UNIVERSITY OF OSLO
Department of Informatics

High level languages on low level devices

Introducing factorized cross-cutting transitions to ThingML

Master Thesis

Jan Ole Skotterud

1. August 2012

Acknowledgements

I would like to thank everybody that has helped me to complete this thesis in one way or another. I would especially like to thank my supervisor **Franck Fleurey** for guiding me and being a patient person. Advice given by **Brice Morin** have been a great assistance in writing this thesis.

Sonen also deserves my gratitude for letting me use all the hardware I needed to build the robot used in this thesis, and never complaining (too much) about all the mess we made there.

To all the great people at the sixth and seventh floor at Ole Johan Dahls house I would like to express my very great appreciation for being a huge resource and aiding me through this thesis by fruitful discussions, breaks, pranks and by being the great people you are. Thank you **Emanuele Lapponi**, **Lill M. F. Reyes**, **Terese Skavhaug**, **Kyrre Haavik Eriksen**, **Kristina Heyerdahl Elfving**, **Åshild Aaen Torpe**, **Bente Bakke**, **Joakim Boarding**, **Simen Heggestøl**, **Gunn Kristin Johansen**, **Tommy Madsen**, **Geirr Sœtre**, **Kjartan Vestvik** and all of you that should have been mentioned here.

I would also like to direct my gratefulness to everybody who have participated in interviews and user tests, without you there would be no evidence for the many claims in this thesis.

A big thank you go to my supportive parents, **Lajla** and **Svein-Erik Skotterud**, who always expected progress and therefore have been driving me forward.

Kine Gjerstad Eide deserves a big thanks for all the love, for supporting me through this work and all the late nights at the university.

At last a big thank you goes out to all the great people at **indiegogo.com** for helping me through the last month of writing.

Without the great support from all of you, this thesis would not have been written.

Thank You
Jan Ole Skotterud

Abstract

Microcontrollers with little available resources, such as program memory, RAM and speed, are in most cases programmed in low level languages such as Assembly, C and C++, these languages can be hard to learn for new programmers and therefore hold them away from microcontroller programming. Because high level languages is easier to learn at a basic level, and more and more non-programmers tend to draw against microcontrollers, this thesis presents a high level programming language called ThingML developed at SINTEF. ThingML can generate code for different platforms, both high and low level, and can run on resource constrained devices such as microcontrollers. In order to find out if ThingML is suitable to become a leading microcontroller programming language, it is compared against three other microcontroller programming languages.

In order to find areas that the users are not satisfied with in today's microcontroller languages, research have been conducted to find areas of improvement to bring on to ThingML.

From the data obtained in this research a suggestion on how to lower complexity in state machines, through the use of *factorized cross-cutting* transitions in general and a implementation in the ThingML language, is presented. The factorized cross-cutting transitions is also compared and evaluated against theoretical state machines, real state machines and a case study is conducted to verify that factorized cross-cutting transitions gives the developers less complexity, better maintainability and overall less code.

What would a compiler compile if a compiler could compile compilers? - *Emanuele Lapponi* 2012

High level languages on low level devices

Jan Ole Skotterud

1. August 2012

Contents

List of Programs

List of Tables

List of Figures

Chapter 1

Introduction

Microcontrollers are everywhere in today's world. It is possible to find microcontrollers in toys, cars, dishwashers and now thanks to Google, in light bulbs [8]. Thus, there is a large demand for microcontrollers; in fact 95 percent of all processors (CPUs) that is produced are embedded microcontrollers [35]. These are resource-constrained units and 55 percent of them are 8 bits with as little as 1KB of RAM and 16KB of program memory available [3]. The production of CPUs have more than doubled from 1999 to 2008 [35], this is a stable upward trend that are filling the world with more and more microcontrollers and it does not seem to stop. Two of the reasons for the blooming growth in the microcontroller marked can be attributed towards cheaper and user-friendlier microcontroller such as the basic stamp [45] and Arduino [4], and the *Internet of Things* (IoT). Microcontrollers is being used to make all sorts of devices able to connect to the Internet and share information [42].

If regular commercial production is looked away from, a broad audience including artists, designers, students, programmers, and non-programmers commonly uses microcontrollers. Most microcontrollers are programmed in C [32], assembly, C++ or C like languages or languages built on C [24, 34, 4, 2], and these languages can be harder to learn for non-programmers than what high-level programming languages can be to learn. Most microcontrollers are in fact programmed by domain experts and highly skilled developers [2]. The reason why microcontrollers are programmed in these low level languages is simply that they have little memory available. High level languages like Java require several megabytes just to be able to run[1] and then the program comes on top. C, assembly, C like languages and languages built on C can compile to bit code that can run directly on the microcontroller and hence requires a lot less space.

There are also other restrictions to take into account while programming microcontrollers. For example, microcontrollers are not suited for processor intensive tasks, as they tend to have only between 4 to 16 Mhz available. This means that the programmer has to think up efficient al-

[1] Java is a language that require a virtual machine (JVM) that can execute and run the Java code

gorithms, or send the data to another unit, which can perform the calculations if processor intensive calculations are needed. Power efficiency can often be important while programming for microcontrollers, some microcontrollers might stay in a remote location without an external power supply for quite some time and thus it is important that they don't run out of power. A good example of this might be a smoke detector that runs on a single nine-volt battery for at least a year. If the developer of a smoke detector had made it do calculations on the side, and send status messages over Wi-Fi, the battery would need to be changed earlier due to higher power consumption. The programming language should handle many of these aspects so that the developer could spend his time on other tasks. Therefore the need for better and easier programming language techniques for microcontrollers is needed, so that the growing community of non-experts and non-programmers can make full use of the microcontrollers full potential. New IDEs and the simpler microcontrollers that are coming to the marked are enabling more and more people to start with microcontroller programming. For example, at the Department of Informatics at the University of Oslo microcontroller programming with Arduino have gained its place in a new course, projects in other courses are more and more often including microcontrollers. The special thing about this, is that these courses are mainly taken by design students, who often don't like programming as much as the students who take the other study programs offered at the Department for Informatics. This shows a growth in non-programmers who set out to learn microcontroller programming.

In this thesis I will evaluate and compare three microcontroller programming languages, were two of them which have large communities and the third one is a new language. The results are important in order to see what a more complete language should offer to the users and how that language should be structured. In order to get input from users, students who are working on different microcontroller projects have been interviewed to find out what they struggle with and what they require from a microcontroller programming language.

Based on these results ThingML [44] is chosen as a candidate that can provide what the users want and need. ThingML is not complete; it is still in development and has some missing features and areas that can be improved.

Two master-students was given training in ThingML, and based on their feedback and my personal experiences, the idea for an extension to ThingML emerged. A potential for improvement in the language in terms of lower complexity, better maintainability and understandability was discovered. *Factorized cross-cutting* transitions were introduced to the ThingML language.

In order to see if *factorized cross-cutting* transitions really improves the ThingML language, it is evaluated against several cases, both theoretical cases, real cases and through a programming exercise. The results from this evaluation show that ThingML and the developers benefits from *factorized cross-cutting* transitions.

1.1 Research Topics

The key topics that is explored in this thesis can be stated as follows:

- To explore the features in microcontroller programming languages and find which features the users needs and wants from a microcontroller programming language.
- To find a suitable microcontroller programming language that meets the users needs and can be a full-worthy candidate to be a preferred microcontroller programming language.
- To find areas to improve in the microcontroller programming language, and improve them to further meet the users needs.

1.2 Motivation

The motivation for writing this thesis comes mainly from two areas, the first area is my own projects and interests relating to microcontrollers and programming. The second is from my role as a teacher assistant (TA) in a design course at the Department for Informatics at the University of Oslo. This course have a great deal of microcontroller programming in it's syllabus. While working on my own projects that make use of microcontrollers, either it is art installation, sensor networks, robots or other things that uses input and/or output devices I often find my self limited by the programming language and need to think out clever ways to do write programs that should be simple. This often is a limitation that in some cases limits what I can build because I can not program the devices and installations to function as I wish.

As a TA for two semesters, in a course which have a lot of microcontroller programming, I get to see a lot of different things, both weird stuff, clever stuff and down right stupid stuff. Through these project it has become evident that there is room for improvement and accumulation within the different native languages to ensure better and easier programming of the microcontrollers, that will enable the students to program their own devices in a more desirable manner.

After spending hours on debugging my own projects and the projects that the students in my TA class works on, I realize that there should be possible to do things differently than what are currently the case. To recap Hoare, I also believe:

> *"(...) that the primary purpose of a programming language is to help the programmer in the practice of his art." [28]*

This is my main motivation for trying to find ways to change and improve how non-experts do microcontroller programming.

1.3 Chapter Overview

The structure of this thesis is as follows: In chapter two the difference between qualitative and quantitative research is presented, the methods used in this thesis is described and some aspects about their validity is discussed. Some important definitions that are used in this thesis is also clarified. From here the thesis is divided into three main parts consisting of two chapters each. The first part deal with existing languages and difficulties that appear while using one of them. Chapter three contains a comparison between three microcontroller programming languages that gives some basis in the selection of a candidate for a better programming language. Chapter four focuses on students who have used the Arduino programming language. They are interviewed in order to find out what they find difficult with the language and which features they would like to have in a microcontroller language.

Part two introduces a fairly new programming language called ThingML that can be used to program microcontrollers and evaluates this language against the three languages in chapter three. In chapter five the most important part of ThingML is presented with examples of usage. Chapter six gives an evaluation of ThingML compared to the languages presented in chapter three. I also evaluate my experiences with the language as well as the opinions provided by two master students who learned to use ThingML. From these data some possible improvements in the ThingML language is suggested.

The third part of the thesis shows one improvement to ThingML explained in detail, the implementation, testing and evaluation of this improvement is presented. In chapter seven factorized cross-cutting transitions is the improvement to ThingML that I choose to work with, the concept is explained with examples. The ThingML implementation of factorized cross-cutting transitions is provided. In the eight chapter the factorized cross-cutting transitions is evaluated against hypothetical examples, a real state machine and a case study.

In the ninth chapter the thesis is summarised with the contributions made from this work and some possible paths to bring this work further is presented.

Chapter 2

Methods

This chapter presents the methods used for data collection in this thesis. They separates the research from my own opinions and keep the data durable and reliable. The methods used to collect data about the users needs, wishes and requirements for a microcontroller programming language and the testing and evaluation of factorized cross-cutting transitions are as follows:

- Interviews
- Workshop / Participatory Observation
- Researchers Impressions and Reactions
- Case Study

The chapter starts with presenting different ways one can approach research and methods one can use. Then some key expressions and definitions used in this thesis are explained to ensure the readers understanding, as well to ensure continued cumulative research within the area of microcontrollers.

2.1 Qualitative and Quantitative Research Methods

Myers [37] writes about the use of quantitative and qualitative research methods for information systems, qualitative research demand usage of qualitative data obtained from interviews, documents and participatory observations for example. To classify qualitative research one common procedure is to distinguish between quantitative and qualitative methods. If two or more research methods are combined it is called *triangulation*. In this thesis, mostly qualitative research methods are used to collect data. Quantitative methods are also used to provide data for the comparison made in this thesis. Justification of the research methods used in this thesis is built upon the reliability and validity principles from Sharp et al. [43].

2.1.1 Quantitative Research Methods

Quantitative research methods where basically developed for use within natural science to study natural phenomenon. Examples on methods that can be used are surveys, laboratory experiments, and numerical methods. The advantage with these methods is that the results can be measured and compared. For example with structured interviews, the subjects are to be treated the same and be given the same questions. The main principle in quantitative research is *accuracy* [37]. The disadvantage with quantitative research methods is that in-depth information is not derived from every participant personal opinion and experience [37].

2.1.2 Qualitative Research methods

The motivation for using qualitative research methods rather than qualitative research method, is what separates humans from the natural world, namely our ability to talk. The strength in qualitative research methods lies in the ability to understand the meaning and context to the phenomenon that are being researched, and the specific events and processes that make up these phenomenon over time in a natural setting [31]. The methods that can be used in qualitative research are for example observation, participatory observation, interviews, questionnaires, and also the researches impressions and reactions [37]. When a researcher or an analyst studies the dynamics of a process instead of statistical characteristics, it may be appropriate to use qualitative research methods [31].

2.1.3 Reliability and Validity

The methods used in this thesis are based on Sharp et al. [43] reliability and validity principles to assess the reliability. The reliability or the consistency of a method is a measurement on how reliable results the method can produce, in separate occasions under the same circumstances. In other words, another researcher with the same methods should be able to get nearly the same results. Reliable results can be expected when structured interviews are used. On the other side, if unstructured interviews is conducted, it can be difficult to replicate the conversation. The validity of the data materials is given by the extent the evaluation methods measures what is relevant for the thesis; both the research method in itself and the way it is conducted. In addition it measures the validity of the environment, an interview subject might for example behave differently if he or she knows that the conversation is being recorded or if it's being filmed than what he or she would otherwise.

2.2 Data Collection for Part one

In the first part of this thesis three microcontroller programming languages are compared.

There are mainly two different approaches for comparing and evaluation programming languages. The first one consist of evaluation the language as a whole and then comparing it to other languages. But by using this approach only benchmark-able fields such as run times get to evaluate [22]. Also the languages are so rich and the measurements techniques so broad so it is impossible to say much more than that one language appears to be better than another [22]. The other approach is to evaluate and compare one feature at the time. As Furuta and Kemp writes:

> *"It is only by varying one factor at the time that the most scientific and replicable results are obtained." [22]*

Therefore this comparison will show which features the different languages have and which languages that are better then the others on the different aspects. This will give some data that can be used later in this thesis. It is also important to get data from the developers who program with microcontroller programming languages. To get an overview of the different problems students who program microcontrollers encounter, the features they would like to have in a microcontroller language and to find out if and how they see solutions to all this, interviews has been chosen as the approach to collect data. Only users of one of the programming languages are being interviewed. It would have been better to interview users of all three languages, but this was omitted in order to save time, and it seems like interviewing users of only one language gave sufficient feedback to map out areas of improvement within that language, which in turn can be brought on to ThingML as well.

2.2.1 Interviews

The interviews conducted in this part of the thesis where semi-structured. The interviews always had a predefined set of questions that where asked one by one, but often the answer could diverge from the questions and the conversation went more freely. The reason for using interviews is that it is possible to learn through the conversation that unfolds. Interviews can therefore be seen as both formal and informal research methods [10]. At the same time, interviews can't be seen as a durable research method alone, because the interviewer only get a summary of how the interview subject have experienced something instead of observing how the interview subject responds in a real situation. Bloomberg [29] writes that interview subjects don't always do what they say claim to do, this can lead to that the interview subjects unconsciously puts them self in a light they want to be seen instead of giving a precise feedback of what they actually do. This could mean that the users encountered difficulties that they did not talk about in order to appear smarter and there could be more areas to improve in the microcontroller programming languages then what is being extracted from the interviews. To make the results more accurate it is possible to take a more ethnographic approach and observe the participants while they program and ask them questions. This was not done since all the projects the interview subjects worked on were in

a near finished state, and it would become very artificial to make them program something in order to reproduce the problems or difficulties that were talked about in the interviews.

2.3 Data Collection for part two

In the second part of this thesis, the ThingML programming language is presented in detail. Then ThingML is being evaluated against several aspects. First ThingML is evaluated against the three languages presented in chapter three, then it is being compared against the results that were extracted from the interviews in chapter four. After this my experiences from learning and using ThingML is presented, this includes things like programming with ThingML and digging through source code and documentation. Two master-students were given the task to learn ThingML and provide their opinion on the language and come with critique and suggestions for improvements.

2.3.1 My Findings

According to Myers [37], the researches impressions and reactions are valid data to use in a qualitative research. Therefore the impressions I made myself while learning to use ThingML and becoming more experienced with the language, can be seen as a source of knowledge that should be taken into account while evaluating the ThingML programming language. It is important that I share my experiences in an as neutral manner as possible and try not to become biased by my own preferences. This is thus hard, but important in order to maintain validity in the results I find.

2.3.2 Workshop / participant observation

Two master-students with a slightly different background were put to the task to learn ThingML. The setting was a hybrid between a workshop, participant observation and learning situation. They were given basic knowledge of ThingML before they started to program on their own. This took four sessions of two hours each. I was always there and in the start of each session I taught them new aspects of the ThingML language, answered questions and gave them tasks. Then they continued to program with me available as a resource in case they needed help. I also observed what they wrote, and asked them questions about what they did and why they chose to do it that way. The observation here is very valuable as it gives insight in how the two students think while they program, and can give a much better image than what reviewing the source code alone or with an interview can give. It is important to recognize that, the fact that I was present and observed the two students can have altered the results, since the students behaviour and experience of the language would maybe have been different if I had not been there the whole time.

2.4 Data Collection for part three

In the third part of this thesis factorized cross-cutting transitions is introduced to ThingML. In order to evaluate that factorized cross-cutting transitions actually offers an improvement to the language, comparisons are done between state machines with and without factorized cross-cutting transitions, both with theoretical state machines, existing state machines and through the programming of a robot.

2.4.1 Case study

To be able to see if the cross-cutting transitions would actually shorten development time, make the programs smaller, less complex, more understandable and comprehensible in a real life situation, a case study was conducted. The two same master-students who have already learned ThingML were participants and received the task to program a robot. Upon completion of this task they where given instruction to factorized cross-cutting transitions, and some training in how to write and use them. Then they were given a new task, namely to program the same robot again, but this time use the factorized cross-cutting transitions. The reason for using a case study at this point can be backed by an argument made by Flyvbjerg:

> *"The detailed examination of a single example of a class of phenomena, a case study cannot provide reliable information about the broader class, but it may be useful in the preliminary stages of an investigation since it provides hypotheses, which may be tested systematically with a larger number of cases."* [20]

However, more studies should be conducted in order to get more data to validate that factorized cross-cutting transitions is easier to write, gives better maintainability and readability of the code and reduce development and maintenance time.

2.5 Terms and expressions

In this section some important expressions that might have several meanings in different contexts is defined so that their meaning should become clear and not cause confusion for the reader.

2.5.1 High level language

In this thesis the phrase *high level language* is often used, and this phrase can have different definitions depending on the context. For example C can be seen as a high level language compared to assembly [41], and then again Java are seen by some as a higher level language than C. One definition of a high level language is: *a language that is easy to use independent of the type of computer* [7][1]. Another definition is that: *many details are handled*

[1] My own translation.

automatic so programmer can write less code to do the same job [39]. A third widespread definition is: *a high level language brings the programmer away from the computer architecture*. In this thesis the C programming language are only considered to be a mid level language, while the four microcontroller languages in this thesis is to be considered as high level languages [41].

2.5.2 Low level devices

Low level devices is another term used through this thesis, and in this context applies to programmable microcontrollers such as the *Arduino* [4] board or the *Basic Stamp* [45]. These boards have a programmable microcontroller and provide a simple way to upload user-generated code, they also provide input and output ports where users can connect a wide range of sensors and equipment. The boards have little memory available, both for the programs that are written for them and other data they might store. They also have little CPU power, usually about 4 to 16 MHz but some newer versions provides up to 72 MHz [23]. Further, these low level devices lack an operating system. Often they only have bootloader[2] that starts up a program that a developer has put on the chip. High level devices on the other hand is devices that are not so recourse scarce and have a higher level of complexity and abstractions such as today's smart phones or even whole computers.

2.5.3 Development for low level devices

Development for the traditional microcontrollers is usually done with languages as *Assembly*, *C*, *C++*, *nesC* and other variations of *C*. The most common way to program microcontrollers, looking away from the industry, is to write the whole program from scratch and maybe using a few libraries found online. This development form spread spread around after the introduction of Basic Stamp and even more after the launch of Arduino. Today most microcontrollers are programmed in C, C++ or Arduino, but some microcontrollers are also programmed in Java[3] and C# with .NET[4].

2.5.4 Hardware Abstraction

When programming in high level object oriented programming languages such as Java for example, the low level functions are abstracted away through the use of classes and methods. These classes and methods wrap the low level instructions in high level method calls and gives code that are much easier to read, write and understand. This means that there is no need to worry about bit shifting, updating the marker on screen during screen printing or splitting up data into TCP packets.

[2] A bootloader is a small program that are burned in to a memory area on the microcontroller chip and starts up a program from the program memory available on the microcontroller chip.

[3] http://www.parallax.com/Store/Microcontrollers/JavelinStamp/tabid/517/CategoryID/13/List/0/Level/a/ProductID/5/Default.aspx?SortField=ProductName%2cProductName

[4] http://www.ghielectronics.com/catalog/product/297/

While programming for microcontrollers and embedded systems it is not only software functions that need to be made easier for humans but also the operation of hardware functions in the code. Computer systems often provide an operating system, hardware drivers and API's to control the hardware, but small microcontrollers does not provide the same for it's user. Therefore the programming languages have to provide that hardware abstraction, and when the hardware abstraction is high or the hardware is abstracted away, the code is not in terms of sending voltage out of a GPIO (general purpose input/output) pin or enabling a GPIO pin to listen for interrupts, but rather being able to say that a light emitting diode should light up or that a LCD (Liquid Crystal Display) should write a defined text.

2.5.5 Events

An event is defined as a something that happens [12] and in the context of embedded systems and microcontrollers, something that happens can be a button that is pressed, a sensor that reads a defined value or a timer that reaches it's set time and fires a message about it's completion. In order to restrict an event from including regular method calls, which indeed is something *"that happens"*, an event has to be restricted into asynchronous calls. So in this thesis an event will be viewed as both software and hardware generated interrupts and asynchronous messages.

Asynchronous message

An asynchronous message is basically a method call, but what makes it different from a regular method call is that the caller can continue to execute statements before the call is completed. Another difference from a regular method call is that a asynchronous message don't need to be a call for a specific method to run, but the event mechanism will make sure that the right event-handler catches the message and acts upon it. A asynchronous message might at some indefinite time return a value, this value is then usually sent as a new asynchronous message back.

2.5.6 Modularity

A module is a collection of functional units, which can be combined into larger applications. Modularity is the design of a system that makes use of modules [36]. In this thesis modularity is a measurement of whether the programming language easily can utilise other software components written in the same language or if it does take a lot of effort to include an already written piece of code into another program.

2.5.7 Parallelism

Parallelism is in simple terms the running of several program flows at the same time. This is a feature normally used in larger processors and computers. Real parallelism is only accomplished by running

different threads on different CPUs at the same time. The resource scarce microcontrollers within the scope of this thesis does not have processors with several cores. Therefore parallelism have to mean something else in this context, and will be redefined to mean several program flows that take turn to execute so that from the outside the behaviour of the two program flows seems to be working parallel.

2.5.8 Statements

In some parts of this thesis a distinction is made between the total number of lines of code and the number of lines of executed code. The executed lines of code is called statements. Every line of code a programmer writes is usually needed in order to make the program function as intended, and the distinction between statements and other code lines is that a statement is an executed line of code, a method declaration, for example, is not.

What a statement are, is best illustrated by an example, in program code 2.1 there is a total of eleven lines of code, but only five statements. The number of executed code is the number of lines inside the methods, and not import statements, or the methods definitions or variables defined outside methods. In order to better illustrate this every statement line have two slashes at the end.

Program code 2.1 Code for the statements code example

```
import "Example.h"
int lifeUniverseAndEverything = 42;
void setup() {
  pinMode(lifeUniverseAndEverything, OUTPUT);    //
}

void loop() {
  digitalWrite(lifeUniverseAndEverything, HIGH); //
  delay(1000);                                   //
  digitalWrite(lifeUniverseAndEverything, LOW);  //
  delay(1000);                                   //
}
```

2.6 Summary

In this chapter the methods used in this thesis have been presented, the methods ranges from interviews, case study, participant observation and workshops. All these methods are placed within the field of qualitative research and the data found and presented in this thesis is qualitative. Much of the data collected in this thesis have been collected with an aim of improving or creating better features in ThingML, therefore an advice from Hoare [28] have always been used as a guideline:

> *"listen carefully to what the language users say they want, until you have an understanding of what they really want" [28]*

By using the above methods the data and results that comes from this thesis can be used in future research to further validate the benefits from using factorized cross-cutting transitions, or extend ThingML or other microcontroller languages.

Some key expressions used in this thesis have also been laid out in order to provide the reader with a clear understanding of what they mean in the context of this thesis.

Part I

Languages Comparison and Indentifying User Needs

Chapter 3

Languages

This chapter gives a comparison of two different languages that are commonly used to program microcontrollers, and one new language that has recently been released to the public. Then these languages will be compared to each other, positive and negative design choices in the languages will be presented if identified. If positive design choices are found, these will be evaluated to see if they should be added to a high level language that can be used to enrich the possibility to program microcontrollers. As for the negative design choices, they are also evaluated to see if they should be in a new language at all or if they should be changed into something better. The languages will be compared in six main areas were the first is the bare minimum of code needed to create a working circuit. The microcontrollers version of *Hello World*, the *blink* program which turns a LED on and off, will be used to do this evaluation. Then the type safeness of the variables in the language will be evaluated, followed by a look into the hardware abstraction levels the languages provides. Then the three last parts will evaluate the ability to handle parallelism, the event handling capabilities and lastly the possibility to use modularity in the language.

The three languages that will be reviewed in this thesis are the C++ modification *Arduino*, *nesC* used in TinyOS and the newly published language *em*. What all these three languages have in common is that they are built on C and translated into native C at compile time. All of the languages are also built to make use of parts of the AVR Libc C library [5] to become compliant with the microcontrollers at compile time.

3.1 Arduino

3.1.1 Background

Arduino has fast become one of the most widespread programmable microcontrollers in the general public after it's release in 2005 [4, 2]. Some of this success can be attributed to the fact that the Arduino board is programmed with a high level language. The Arduino board is based on *Wiring* [48] and is programmed with an extension of the C++ language

called Arduino[4]. *Arduino* consist of special libraries and functions designed to make it easy to use the hardware that is compatible with the Arduino board. Even though the Arduino programming language is high level, the hardware is not abstracted away and the API provides simpler functions to interact with the hardware [2].

3.1.2 Language Construct, the bare minimum

A fully functional Arduino program only needs the implementation of two functions; *setup* and *loop* [4]. The *setup* function contains mainly two things; declarations that only needs to be set once, and the definition of what specific *pins* that will handle input and output. The *setup* function will only run once the Arduino board boots up. The rest of the program is put in the *loop* function or in custom written functions and calls from *loop* or *setup*. Global variables are written outside the scope of the functions. A C++ program also needs a main function in the program, but the compiler adds this function with calls to *setup* and *loop* during compile time. Arduino also support more complex programs with the use of classes, this makes it possible to achieve object oriented programming (OOP) functionality with the Arduino. Arduino have a rather short implementation of the blink example as shown in program 3.1.

Program code 3.1 The Blink program written in Arduino.

```
void setup() {
  pinMode(13, OUTPUT);
}

void loop() {
  digitalWrite(13, HIGH);
  delay(1000);
  digitalWrite(13, LOW);
  delay(1000);
}
```

It is a positive feature that the amount of code needed to get basic functionality up and running is low, this allows novices to get a feeling of mastering the language in a short time and possibly inspiring them to keep on going.

3.1.3 Variables and Type Safeness

Arduino provides the same basic set of data types as C provides. In addition it provides the String data type, this is simply a char array in a nice wrapper, inspired by modern programming languages. In version 19 Arduino also provide a class encapsulation of the String data type. The type safeness is working on the object level where casting are checked at compile time. When it comes to low level variables, nothing is checked during compile time and it is possible to assign a *long* into an *integer* with a possible loss of data and no casting required.

The Arduino language does not support dynamic typing[1], and lack good restrictions for this on low level variables. Arduino make use of global variables and that could be considered as a negative programming practice as global variables can be hard to keep track of when the program becomes large and complex.

Integers are used to map pins for input and output on the Arduino board, but as the Arduino language does not provide a check to make sure the provided integer can map to an actual pin on the board. This means that when set to read a specific value from for example pin 1023: `analogRead(1024)`, it could return a reading without any meaning.

3.1.4 Hardware Abstraction

As mentioned earlier the Arduino programming language are high level but the hardware is not abstracted away. The API provides simple functions to interact with the hardware [2]. Some examples to this are the *digitalRead* and *digitalWrite* functions that can read and write to the digital ports on the Arduino board and interact with connected hardware. In other words, high level functions is used to interact with low level hardware. For example LEDs and piezo elements, or even more complex equipment such as LCDs. Arduino also provides simple APIs to read and write over the serial port, to enable debugging and easy communication with a computer and external programs. These APIs that provides high level abstractions of low level operations makes the Arduino an easy platform to learn and attracts new non-experts users [2] to the field of micro controllers.

3.1.5 Parallelism

One thing that the Arduino implementation doesn't support natively is multithreading applications. Multithreading could give benefits in embedded systems and make it easier to create more complex Arduino circuits and programs. Since Arduino supports external libraries and classes, people from the open source community have written thread implementations that can be use with Arduino. An example is a multithreading implementation [46] that gives a full non-preemptive multithreading functionality on the Arduino board. Using this library, however, will probably take up much of the available memory on the Arduino board, because the library consist of 17 files and over 1800 lines of code. As a second alternative there exists a *TimedAction* [1] library, that gives a thread like functionality. This library will make chosen parts of the code run at certain time intervals, it's not the same as threading, as it will not allow switching from one code scope to another without finishing an ongoing scope first. Rather this alternative will first run one part of the code, and then run another part of the code when a predefined amount of time has passed.

[1] Dynamic typing is when there is no requirement to tell the compiler which data type the variable will hold, and the data type in the variable can be changed during run time.

3.1.6 Events

Event-handling is important to embedded microcontrollers and Arduino have support for hardware interrupts as well, but on most types of Arduino Boards there are only two pins[2]that support hardware interrupts. Then again it can be configured to listen for changes, rise or fall in voltage or if the voltage on the pin is low. If the selected event to listen for occurs, the code will jump to the specified code and run this before returning to the place it left off in the program flow. The Arduino language does not have support for software-generated event without the use of a third party library or event-handling system written by the user.

3.1.7 Modularity

The Arduino platform has a large community thanks to it's open source offering, the amount of supported hardware and ease to learn. This also means that a lot of code is shared online and are available for everyone to use. Since the Arduino platform doesn't have a framework that support modularity or facilitates reuse of contributed code, copy and paste is the common way to share code. The API provide an abstraction of the hardware, this means that novices almost only use what they find online with a small amounts of modifications, and only experienced programmers manage to fully take advantage of the code and modify it to fit their precise need[2].

3.2 TinyOS and *nesC*

3.2.1 Background

TinyOS is an event driven operating system and it is specially designed with sensor network nodes in mind [24, 47]. It is very energy efficient and has implemented libraries to many hardware communications units such at Wi-Fi, bluetooth and radio communication devices [13]. TinyOS is also designed to take up as little space as possible after the code is compiled; this is because microcontrollers tend to have a very limited amount of available memory. Further, TinyOS is implemented in a language called *nesC*[34, 24] which is an extension to C, the same way Arduino have its own extension to C++.

3.2.2 Language Construct, the bare minimum

nesC is programmed a little different C, C++ or Java, the syntax is basically the same as C when it comes to declarations and initialisations [34], but the rest is rather different. In order to write a *nesC* program two things

[2]The number of pins that can handle input events (or interrupts) is defined by the AVR chipset that is being used in the microcontroller, not by the programming language itself. For example the chip used in the Arduino Mega board (ATmega1280) has six hardvare interrupt pins.

are needed, a module and an interface [24]. The module is used as a component, for example as an sensor component, and the interface is used as an configuration of that component. The configuration is also used to wire several components together. The nesC blink program example (program 3.2) shows the configuration and the implementation file that are needed.

Program code 3.2 The Blink program written in *nesC*. The configuration file and the module file

```
######## The configuration file #########
configuration BlinkAppC
{
}
implementation
{
  components MainC, BlinkC, LedsC;
  components new TimerMilliC() as Timer0;

  BlinkC -> MainC.Boot;
  BlinkC.Timer0 -> Timer0;
  BlinkC.Leds -> LedsC;
}


############ The module file ############

#include "Timer.h"

module BlinkC @safe()
{
  uses interface Timer<TMilli> as Timer0;
  uses interface Leds;
  uses interface Boot;
}
implementation
{
  event void Boot.booted()
  {
    call Timer0.startPeriodic( 250 );
  }

  event void Timer0.fired()
  {
    call Leds.led0Toggle();
  }
}
```

A *nesC* program might be difficult to learn for inexperienced programmers, and it also requires a different way of thinking than experienced C and C++ programmers are used to [13]. This is because *nesC* is build as a ”non-blocking” and ”split-phase” [34, 24] language. This is the same principle as with asynchronous messages and means that when a call is made to for example a *sendData* function, it returns almost right away, even if the data isn't sent yet. This means that the program flow can continue asyn-

chronous [24] and at some indefinite point of time, TinyOS will call a call-back on the program, for example *sendDataDone* give feedback saying that the data are sent and calculations dependent on that transmission being done can be performed. Since TinyOS runs like this it means that it would be useful to understand threaded programming in order to write good and efficient *nesC* programs. Because of the complexity that can arise in a *nesC* program that have several modules wired together, the user-base mainly consists of highly skilled programmers.

3.2.3 Variables and Type Safeness

nesC contains all the data types that are found in the C language, and does not provide any other custom data types, but programmers are free to create custom data types. The *nesC* language does use global variables, thus only within their own namespace. A namespace in nesC is typically restricted to interfaces, which means that modules that implements the same interface is within the same namespace [24].

3.2.4 Hardware Abstraction

The *nesC* language provides hardware abstraction to the developers, this abstraction inferes the need to write configurations that binds the program elements to the hardware elements. The abstraction causes configurations to be more written almost more often then the modules in the programs *nesC*. Modules are built on top of sets of abstractions, which are then encapsulated in configurations [33].

3.2.5 Parallelism

nesC is a language that is designed to be running in a threaded mode, so it offers great support for multithreading. The basic working of messaging in this language is based on threading and whole applications are run in a split-phase non-blocking mode. This is made possible by the bidirectional interfaces that the language provides, which in turn allow modules to both send and receive messages within the same interface. This support for multithreading enables the developers to write advanced programs.

3.2.6 Events

Event handling is a main component in the *nesC* language, this goes hand in hand with the multithreading capabilities. *nesC* provides both software events as well as hardware events. Software events will break the program flow, and the event code will run to completion, unless there is a hardware event, then this event will have precedence over the software event and the hardware event will run to completion before the application jumps back to where it left of.

3.2.7 Modularity

nesC is also designed for modularity and reuse of code [24]. The *nesC* language provides a module based implementation. Once a module is created for a program, this module can be reused in all programs that uses of the same sensor or device that the module was implemented for. Because of the interfaces the language provides, it is rather easy to sew several modules together and make them communicate in new applications without too much tweaking. The interfaces in *nesC* are bidirectional and modules can both implement them and use them. The interfaces the module implements contain method signatures and the module must implement these methods. The interfaces a module use, must also be implemented in other modules that are supposed to communicate with mentioned module.

3.3 EM

3.3.1 Background

em is a newly developed language which was published in an article and a doctor thesis in August 2010 [2, 3]. The *em* language was developed with specific focus on three areas. Firstly, that the language would implement modularity that allows portability and interchangeability [2]. Secondly, em was designed to generate code that would take up less space on the microcontrollers than other widely used languages. The third focus area was that the generated code would run faster and more efficient than commonly used micro controller languages does.

3.3.2 Language Construct, the bare minimum

em was developed by extending the C language with approximately 30 keywords and new syntax patterns [2]. This was done to make it possible to have a strict form for inheritance in the language. The developers also removed the support for global variables and functions. This is done to prevent concurrency [3], and language pollution. The creators of *em* implemented a more scripting language like syntax to keep the code as short as possible. They took away the need for semicolons at the end of lines and declarations [2, 3], this choice was made to lower the number of typing errors, and also to attract new programmers who are used to program in scripting languages[2, 3]. As seen in the *em blink* example program (program 3.3), there is only need for one file for configuration and implementation. The configuration in em might be hard for a non-experienced programmer to grasp at first.

Em uses modules as the main code component. A module encapsulates functions and the data it implements. This is called the modules *features*. Modules in *em* can implement interfaces, and if they do they must implement all of the methods that are defined in this interface [2]. This is similar to the way interfaces are used in Java. The modules can have two

Program code 3.3 The Blink program written in em.

```
from em.bois import EventDispatche
from BoardC import Led
from BoardC import TimerMilli0

module BlinkP {
  config rate: UInt16
}

private {
  var blinkEvent: EventDispatcher.Event
  function blink( event: EventDispatcher.Event ) void
}

def em$configure(){
  rate = 500
}

def em$construct(){
  blinkEvent.initOnHost(blink)
}

def em$run(){
  TimerMilli0.start(rate, true, blinkEvent)
  EventDispatcher.start()
}

def blink( event ){
  Led.toggle()
}
```

different kinds of features, private and public. The private features are only available to the parent module. But the public features are freely available to other modules. Modules that access other modules features are called users [3].

In *em* a new language construct called *proxies* are implemented. Proxies are used to bind modules together without the modules having to know about the other modules implementation of functions [2]. The proxies connect the modules and their interfaces in a way that allows the modules to communicate, and used with composites, proxies give *em* a great strength that few other languages have. Proxies can implement some methods from a interface, and if several different modules also implements the same interface as the proxy, but don't have defined the functions that the proxy have implemented, the module will use proxy implementation instead. This is useful if, for example several program modules are intended to broadcast a message in a specific form over an output pin on different events.

3.3.3 Variables and Type Safeness

The *em* language provides the same data types as the C language does and doesn't implement any new data types. The only difference from C is that in *em* the width[3] of the *primitive type* must be specified, if it can vary [3, 2]. If the data type can be signed or unsigned this also have to be specified. An integer may be declared like this:

```
var myInt :  UInt8 = 42
```

Em also support use of pointers the same way they are used in C++ but the developer argue that it is not necessary to use pointer due to the power given in the parameter passing semantics in *em* [3].

Another design choice the creators made was how to declare variables. Since there are no global variables, all variables are declared inside scopes inside the modules and there is also a special syntax to declare the variables. A variable start with the keyword "var" followed by the variable name, the variables data type and optionally an equal sign and a value.

3.3.4 Hardware Abstraction

em provides the developers with a hardware abstraction layer (HAL) that make it easier and faster to develop programs that are independent of the low level hardware on the micro controllers. *em* use interfaces to define the HAL that defines the basic functionality of common microcontroller functionality such as general I/O pins, timers, serial ports and other integrated peripherals [2].

[3]The width of a variable refers to the amount of space the variable takes up in the memory, typically this is eight, 16 or 32 bits.

3.3.5 Parallelism

The *em* language does not provide parallelism or multihreading. While developing the language one important thing was that the language should use as little memory as possible. Implementing threads would have taken a great portion of memory from the micro controllers. While using threads each thread needs it's own stack in the memory and there is a need to implement various support mechanisms such as locks. According to the developers of *em* the amount of memory this would take from the microcontrollers, overweighted the benefits from implementing the threads [2].

3.3.6 Events

The *em* language supports event handling, both hardware events and software generated events. When an event is created either from hardware or software, it is sent from the corresponding module to an event dispatcher. The event dispatcher can send the message, or trigger a function, when it is configured to do so. The configuration can happen either in compile time, as in the blink example (program 3.3), or it can be configured dynamically at runtime, this feature allows the program to change at runtime and is a positive feature but also a potentiality pitfall.

3.3.7 Modularity

Modularity is a key concept in *em*, and the language was built around the idea of reusing code with as little effort as possible. And then not only between microcontroller programs, but also over platforms. This means that it should be possible to use the same module code on an Arduino board as on a Basic Stamp [45] board, the only thing that is needed is some customization in the ”board” import sentences, and possibly in the configuration part. But the code regarding I/O pins and communication with other modules can stay the same. The modularity in *em* is also enhanced by the use of interfaces and proxies that can bind modules and interfaces together.

3.4 Language Comparison

In this section the three languages that are presented in this chapter will be compared to each other on the six areas that are laid out above. Thereafter it will be given an evaluation of how difficult the different languages are to learn based on a short interview. The interviews are conducted on informatics students conducting their bachelor degree. All of them have experience with Java and C, but in a varying degree. Some of the participants have experience with the Arduino microcontroller, and the remaining participants have no microcontroller experience at all. The programs that are used in the interviews are the *Blink* program which is the equivalent to *Hello World* in the world of programmable electronics.

The *blink* program implementation is provided in program 3.1 on page 18 for Arduino, program 3.2 on page 21 for *nesC* and program 3.3 on page 24 for *em*.

The minimum amount of lines of code that is required to write the *blink* program is absolutely the least in the Arduino implementation. It consists only of nine lines of code where five of them is statements. *NesC* proves to be the most complex one in this example. It requires two files, one for configuration and one for the module itself. All in all it consists of 29 lines of code where ten lines is statements. *em* is found in the middle with 23 lines of code where eight of them is statements.

Table 3.1: Blink example comparison

	Arduino	em	nesC
Lines total	9	23	29
Statements	5	8	10
Number of files	1	1	2

This shows that the *Arduino* language requires less code to get a simple circuit up and running. But in a larger and more complex example one of the two other languages could be the one that requires the least code. This is because of *nesC* and *em's* module based approach and ability to reuse code components.

Strict variable types can make ting easier or harder depending on what kind of programming language the developer are used to. If the developer is used to scrip languages such as JavaScript [30] or PHP [40] the notion of using one variable to hold different types of data is known. This could make it easy and strait forward to program with no concern about using the right data types. Then again it could cause problems if the control over the assigned variables and data is lost. The three languages that are evaluated in this dissertation all provide strict enforcing of variable types, as a high level language should. All three languages provide the data types that are present in C, and *Arduino* also provides *Strings*. In *Arduino* and *nesC* it is possible to cast a variable from one data type to another e.g. *char* to *int*. This is because all data types are built of a sequence of bits. The data types may vary in length, this can cause a data loss while casting from one data type to another. Werther if *em* also supports casting between data types is unknown as it is not mentioned in either the article [3], or in the PHD dissertation [2]. The *em* language does provide a *cast-expression*, it is not explained what it is or does, but the name implies that it have to do with casting. Variable checking could pose a problem especially as shown in the *Arduino* section. It is possible to read and write to I/O pins that might not exist, this could lead to confusion an long hours of debugging for the developers. The I/O pins used in a high level microcontroller programming language should be validated in order to prevent this problem.

All three languages provides hardware abstraction. The *Arduino*

language is the language that have taken the abstraction the furthest. It provides high level functions on all the low level operations that is needed on a microcontroller. *nesC* and *em* does this too, but because of the way modules must be configured while programming in these languages it gives a feeling of programming in a lower level of abstraction. *Arduino* and *nesC* provides the users with the possibility to program their microcontroller on a low level if it is desired. In order to fully make use of this more in depth knowledge is required and it could give some benefits. For example accessing the port registers directly will save several clock cycles in execution time and take up less program memory on the microcontroller.

Table 3.2: Hardware abstraction comparison

	Arduino	em	nesC
Hardware abstraction	x	x	x
Low level programming	x	0	x
High level feeling	x	0	0

When it comes to parallelism only *nesC* have implemented native support. The developers of *em* argued that supporting multithreading would take up to much space compared to the gained benefits it would get from implementing multithreading capabilities [2]. This is thus the developers opinion and the possibility for multithreading could have been included and users who need this functionality could been allowed to use it. For the *Arduino* platform there are third party developed libraries available that enables multithreading capabilities, but these are large and would eat into the available microcontroller memory.

Table 3.3: Multithreading support comparison

	Multithreading Support
Arduino	Third party library
em	Not at the current time
nesC	Natively

All three languages support hardware events and will cause the program to deviate from the current flow and execute code registered for the events, if it receives a hardware interrupt, in real time. *Arduino* is the only one of the three languages that don't support software events. *em* and *nesC* supports this because of the way the language are defined with modules and interfaces that natively allows a wider range of communication between program components. *em* supports software events through its "`EventDispatcher`" and can trigger software events at given intervals or events. *nesC's* software events support are strongly supported by its multithreading capabilities and can execute events without interfering with running modules.

Modularity is important in order to reuse previously written code and code written by other developers. The *Arduino* platform does not have a framework that supports modularity or facilitates reuse of code. All reuse of code happens via copy and paste, or by writing own libraries that can be used at a later time. *em* and *nesC* are built upon a principle of modularity and it is a key concept to the way programs written in this languages are created. When one module is configured it can be wired to another module without being configured again.

Table 3.4: Modularity support comparison

	Arduino	em	nesC
Supports modularity	No	Yes	Yes
Facilitate reuse of code	No	Yes	Yes
Module based	No	Yes	Yes

As seen from table 3.4, *em* and *nesC* are both strong in terms of modularity and code reuse while the *Arduino* language have nothing to offer beyond copy and paste of code.

As shown in the *blink* examples, *nesC* and *em* have a rather similar syntax and are different from the *Arduino* code style. *em* and *nesC* syntax might be harder to learn than the C++ implementation that Arduino uses. The *Arduino* language is, after all, designed with non-programmers in mind [4]. Then again, the idea of using modules and interfaces as the building blocks of an application are smart in terms of code reuse and portability. To find out if this is the case, several bachelor students in informatics were presented with the source code to the *Blink* program in the three languages. They were asked if they could find out what the three programs did. After they had taken their guess, they were told that the program would make a LED turn on and off. Then they were asked what language seemed to be easiest to understand, which language seemed to be the easiest to learn and which language seemed to be best fitted for reuse of code. The students that were presented with this code were both students with microcontroller experience and students without micro controller experience, everybody was experienced with Java and C.

The students that didn't have any experience with microcontrollers had a hard time grasping what the programs actually did, but after they were given an explanation stating that the programs are supposed to turn a LED on and off, they understood the code better. The students with microcontroller experience understood the code, they had all used *Arduino* before, so they knew the *Arduino blink* program right away, but it took some time to understand the other two *blink* examples. Here it can be argued that they stated to understand the *nesC* and *em* code because of the use of the "led" names in the code, and they drew a conclusion that the programs did the same.

When asked which language that is easiest to understand most of the participants answered the *Arduino* code, but a couple of the

microcontroller-experienced students said that the *nesC* also made quite a bit of sense. No one liked the *em* code syntax, even though the code appears to be cleaner than *nesC* code. When it comes to ease of learning, all the participants stated that the Arduino language semed to be easiest to learn, but this might be because Arduino had the shortest implementation of the *blink* program. Still, this can't be stated with certainty. In terms of modularity the participants primary gave two answers; the non microcontroller experienced students stated that the *Arduino* language (probably because it had less code) is the most modular language. The microcontroller experienced students thought that *nesC* were better suited for modularity.

Table 3.5: Result of the student interview. 1 is best and 3 is worst [4]

	Arduino	em	nesC
Ease of learning	1	3	2
Easy to understand	1	3	2
Suitable for modularity	2	3	1

3.4.1 Discussion

As stated by both the comparison and the student interviews, *Arduino* is the language that requires the lowest amount of code in order to get started with a small project. This causes it to seem easier to learn and more manageable for new programmers and non-programmers. This is an important feature of any language designed to capture a broad audience, and it is a trait that should be present in a high level language for microcontrollers.

Enforcing strict typing of variables is important and will prevent many possible errors and possibly lower debugging time in compare to having loose variable types. Strict typing is particularly important while programming for microcontrollers because it gives possibility to select exactly the variable size that is needed and this can help saving program memory on the microcontroller, contrary to using a 32 bit variable, or larger, for all data types. Strict variable types are standard in many high level object oriented languages and should also be a feature in a high level programming language for microcontroller. This will remove the possibility to use wrong data types in a variable in wrong places and ease the learning and debugging process for novices.

Variable checking is important when it comes to assigning I/O pins and it should not be possible to read or write on a not-existing I/O pin. This restriction should be implemented in high level microcontroller language

[4]It is important to remember that this interview was performed on a small sample of students, and their understanding of modularity might not be correct. This can also be seen by the fact that em was rated as a less modular language than Arduino. While in fact, em is more modular than the Arduino language.

as a safety measure. There should however be an override mechanism in place in case the developer have got his hand on a microcontroller with pins that are not yet available in the language. There are cons and pros for implementing parallelism in a microcontroller language. *nesC* has good support for multithreading and achieves to have programs with asynchronous message flow and event handling, this is a positive feature because it allows for different and more complex program structures than *Arduino*. On the con side there are memory constraints. Multithreading takes up a lot of memory, and was not implemented in *em* because of the amount of memory it would take. *Arduino* only provides it as a third party library, which is quite large. Microcontrollers manage to function properly without multithreading capabilities but it should be possible to have multithreading capabilities in a high level language, and rather let the user choose to use them. A the memory cost should also be taken into consideration.

Event handling is obligatory for any microcontroller language. It wouldn't make much sense in having microcontrollers if they could not react to input created by external entities. Just imagine a smoke detector that detect smoke, but can't trigger an event to raise an alarm, it would be useless. The same goes for all kinds of external input. In short, event handling is essential and must be included in a high level programming language for microcontrollers.

Modularity is important to save work and reduce the amount of code that is being written over and over again. The way *nesC* and *em* provides modularity seems hard to grasp for little to non-experienced programmers and is not necessary the right way to go to provide modularity and facilitate code reuse. High level languages are often object oriented and uses classes, classes are highly reusable but often only have methods for retrieving and setting variables. Not for sending them. This could be improved with a stronger use of interfaces and a message dispatcher module similar to the *EventDispatcher* available in *em*. Modularity should be a key concept in a high level programming language for microcontroller, but in what way it should be implemented is outside the scope of this thesis.

3.5 Summary

In this chapter, three microcontroller programming languages have been reviewed. They have been evaluated in six different areas, and compared to each other on these areas. There have been found positive and negative traits in all three languages, and some of these traits have been evaluated as good enough to be brought into a high level programming language for microcontrollers. These evaluations are based on several aspects, like how easy it is to learn, will it consume too much memory and will it reduce the amount of code that needs to be written. If the findings presented above are possible to integrate in a high level language, microcontroller programming may become easier, faster and less error prone then it is today.

The traits identified in this chapter that should be in a high level microcontroller programming language, can be listed as follows:

- Strict variable typing
- I/O pin validation
- Hardware abstraction
  - With the possibility to write low level code
- Parallelism or parallel like behaviour
  - Support of asynchronous messages
- Provide modularity and facilitate for reuse of code
- Should be easy to learn and understand
  - Which, as stated by the students, means a low amount of code

Chapter 4

User Interviews

In order to make a programming language successful, it is important to take the users requirements and needs into account, as it is the amount of people who uses the programming language determines its success. In order to get a large users base, the language have to be easy to understand, easy to use and it must give the desired results. To identify aspects that could be useful to have in a micro controller programming language and aspects that should not be there, interviews were conducted with students who had experience with microcontroller programming at the University of Oslo. Only students with *Arduino* experience were chosen because *Arduino* is one of the fastest rising platforms [2]. At first the plan was to arrange workshops for students with little or none *Arduino* experience as well, but after two sessions it became quite clear that they did not learn enough during these workshops for it to be effective enough regarding this thesis and the workshops were discontinued. In the next section, it is laid out how the interviews were conducted, the projects the different students worked on are presented, and then the problems that occurred in these projects are laid out. From this information there is extracted some features that should be part of a microcontroller programming language in order to ease the development process for the users.

4.1 The Interviews

The students that were selected for this study have all participated in some kind of project that made use of an *Arduino* microcontroller and the projects ranged from simple quiz machines with three buttons to large autonomous flying balloons with several sensors. In the interview the students where first asked questions on what kind of knowledge they had with microcontrollers, what kinds of projects they have done with microcontrollers and problems they have run into. Then they where encouraged to talk about how they solved the problems; both programmatic or if they solved problems by using different hardware. They were also encouraged to provide the source code for one project, and talk a little about this code and point out parts they where particularly happy with and parts that might not work as intended to. After this

they could propose changes to the language that would have made their development process easier. The interview is available in appendix A. The questions (available in appendix A) was used as a guideline in the interviews, the interviews was always casual and the talk often diverged from the question and went freely.

4.2 Results

The students with microcontroller experience had only worked with the *Arduino* microcontroller. Most of these had learned and only used *Arduino* in a University course. Only a few had prior experience with *Arduino*, and not all of them had continued to use the *Arduino* microcontroller after the completion of the project.

Table 4.1: Distribution of students who took a course with microcontroller programming, students who had experience from before the course and students who continued to work with microcontrollers after the completion of the project.

Experience before course	3
Experience from course	7
Continued with microcontrollers after course	7
Total participants	10

4.2.1 Projects

The students participating in these interviews have been working on several quite different projects. Some of them bigger then others, but they all have one thing in common, namely that they have worked on the projects outside of the course. Some of the projects started as a course project but where continued after the course had ended because the students either wanted to see how far they could take it or they had been encouraged to continue develop it for the university.

Range Finder

This robot uses a two-wheeled base and a h-bridge[1] *Arduino* shield, and could move around freely. It has a distance sensor that works like a sonar. It sends out a signal and then measure the time until it gets back, then the robot know the distance to the objects around it. This sensor was then attached to a servo, which paned from left to right with about a 130-degree radius. Then the robot combined the data about the direction the servo pointed the sensor and the distance the sensor read, to make a decision on when to change the driving direction. Another feature the robot has is that

[1] A h-bridge is a circuit that makes it possible to control the speed and spinning direction of a motor without changing the wiring.

the distance measured to a object can be shorter on the sides than if the obstacle is present in front of the robot, before the robot need to take a turn action.

Line tracker

This robot is also quite simple, but have a potential of becoming much more complex with relatively little effort. This is a line following robot, which uses three infrared lights and three infrared transistors to detect if the robot is following a black line or if it is deviating away from the line. The robot used a two-wheeled base, controlled by a standard h-bridge shield for *Arduino*. The robot makes decisions about the direction to drive based on the values that are read in by the three infrared transistors.

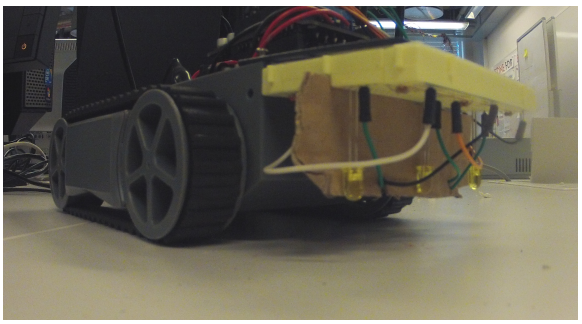


Figure 4.1: The Line Tracker

Quiz Machine

One seemingly simple project is the Quiz machine. The machine displayed quiz statements and questions on a screen and they could either be true or false. The user have to answer the questions by pushing either a red or a green button and get a score depending on correctness of the answer and the time used to answer the question. From a hardware point of view it only consisted of three buttons and an *Arduino* board, one button for true, one button for false and one button for reset. The *Arduino* board just listened for input on the buttons and send signals to a computer that runs a processing sketch which showed questions, points, images and sound on the computer screen.

Figure 4.2: The Quiz machine in action

Loopy

The next project in the case study is something completely different from the rest of the projects. Loopy is a music controller table, which enables users to create music by changing colors on sixteen different wheels. The wheels are arranged in a grid of four by four, and each wheel is divided into five areas with five different colors. Inside the board there is mounted one light emitting diode and one light resistor for each wheel. The amount of light that reflects from the wheel is enough to determine the color that is chosen. The combinations of the chosen colours decides the music that is playing. To accomplish this a *Arduino Mega* board is used with sixteen light emitting diodes and sixteen light resistor. The *Arduino* card communicates with a PC that reads the values and triggers the right sounds. The computer also runs a Processing sketch that displays graphics, which corresponds to the music that are playing.

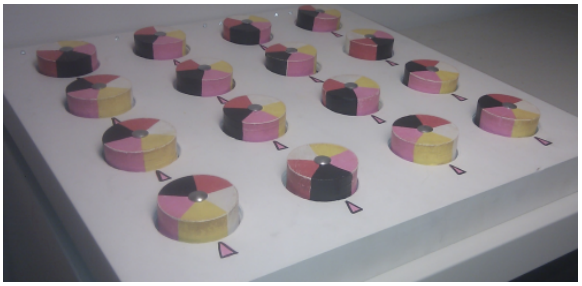


Figure 4.3: The Loopy table

Blimp

The last project in this case study is a fully automated Blimp[2], this Blimp has five range sensors, PID controller, three motors and two servos, all controlled by a *SeedFilm Arduino* clone. The Blimp uses the range sensors to navigate safely, it usually goes straight forward, but if the Blimp detects that it is approaching objects from either the front or from the sides it changes its directions. If it senses that it is getting to close to the roof it corrects its height. The height is controlled by a servo that turns two of the motors to face upwards, downwards or to the back, and those two motors control the thrust. The direction of the

Figure 4.4: The Blimp flying autonomously during a demonstration

[2]A Blimp is a airship with a gondola underneath.

heading is controlled by a servo that turns the tail fin and a motor that goes faster or slower depending how much and fast the Blimp have to turn.

4.2.2 Challenges

This section presents the challenges that the students met in their projects. Solutions and possible solutions to these problems are also presented in this section. All of these solutions or solution suggestions also derives from the interviews.

Quiz Machine

The difficult part in this project was to listen on all three buttons concurrently and react to the different input. To cope with this problem the students changed their *Arduino Uno* card to an *Arduino Mega* card which have six interrupt pins compared to *Arduino Uno's* two interrupt pins[4]. The students working on this project reported that the difficult part was to listen to several buttons at the same time and chose to use a microcontroller with many interrupts instead of checking one and one button directly in the code since this would be unfair if both buttons where pushed at the same time, then one of them would get precedence over the other. While using interrupts this would not be an issue since the interrupt always trigger the code, of the first button pushed, first. The code corresponding to the second button push would then execute secondly. If they were to write the code them self one button would always be checked before the other and the game would be unfair. The checking of the reset button would also take time and hinder a potentially ”answer push” to be registered.

Line Tracker

The main problem with this robot was the control structure: it was difficult to detect and decide what would be the most efficient way to arrange the *if* tests that would decide if the robot should travel forwards, backwards or turn to any of the sides. It became especially hard when the robot detected black areas under two or three of the sensors. To clarify this, imagine a black line going straight forward that the robot follows, then the line splits both left and right and the robot finds a T-junction. Now all three of the sensors detect the black line, and the robot could possibly go forwards, left or right (illustrated in figure 4.5. Forwards obviously being a wrong decision in this example. Another example is if the line only diverges in one direction, both the middle and one of the other sensors will detect the black line and the robot could go either forward or to the side. And of course the lines may diverge with less then 90 degrees and so on. In the case of this robot, the need for a easier way to write the control structures that decides the actions of the robot is needed.

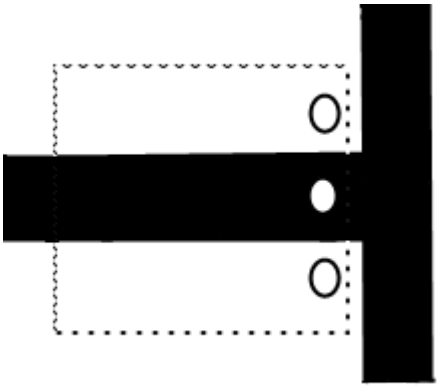


Figure 4.5: The line the robot is following is the black area. The robot is outlined with the dotted line and the three infrared sensors are marked with circles. The decision problem rises when all three sensors detect a black area at the same time.

Range Finder

The Range Finders biggest problem had to do with the update rate of the range sensor, the creator of this robot had an intention that the robot should be able to sweep the sensor from side to side and find the range to close objects at all times. The approach he chose, was to give the servo coordinates, wait for it to turn, do a measurement and then give the servo new coordinates, take a new measure and continue like that. One of the biggest issue in the code was that the range returned by the range sensor had to be explicit checked before the robot could turn the servo and do a new measurement and calculate the turn rate depending on the objects that could appear. He would like to do it more dynamically and simultaneously and get a signal each time the range sensor returned a measurement of something being close, and get the servos orientation with it so that an appropriate turn vector could be calculated. All in real time without the need to wait for the results.

Loopy

This project has encountered a lot of problems among the way, many which are related to hardware but also some to software. To mention one of the hardware problems, the photo resistors drained too much power from the *Arduino* board, causing the values read from the sensors to sink over time and thus invalidate the calibration and makes the board play the wrong sounds. A software related problem was caused by the amount of communication between the computer and the *Arduino* board causing requests for sensor readings to be dropped because of time-outs. The reason for this approach was that the main application in this project was programmed in Java. The *Arduino* sketch did nothing more then listen to Serial communication and read and write to pins as instructed to, and return the values. The package drops in the communication could have

been avoided if the *Arduino* card was programmed to automatically read the values and only send them to the PC if the values had changed more than a given threshold, but this proved difficult since the board had to be calibrated with different values for the different colors in different rooms with other lightning conditions.

Blimp

This project has a more involved source code than the other projects presented in this chapter, and it stretches over 700 lines. This large source code did cause some trouble for the developers because it was hard to find the right segments of code to edit while they were debugging. This is a problem mainly because of how the IDE (integrated development environment) works. The biggest problem encountered during this project, if hardware problems is excluded, was that everything happens sequentially. For example if a sensor detected something that implied an action to execute, the other sensors became blocked while this action was executed. Lets for example say that the blimp had flew in between a lower roof segment and a bookshelf that was located in the middle of the room. Then the sensor on top would register that the blimp needed to be lowered, and while the servos turns the rotors upwards and the motors starts to run, the blimp should have detected that there is a obstacle underneath, but it can't detect that before the process of diverting the roof have been initiated. This can't be done because of the lack of parallelism. They had a strong need for it to be possible to use threads on the microcontroller so that each sensor could run on it's own thread and give notifications while actions needed to be taken without the actions have to block the other sensors from doing their work. They also wanted a round robin implementation in the language where it should be possible to register sensors and the system would get input from the sensors while the rest of the program could run without stopping up to listen for sensor input. They also wanted for better encapsulating in the language.

4.3 Findings

From these five projects it is possible to extract four main areas that the students had trouble with and saw a need for improvement. The first area is interrupts. Real hardware interrupts can only be provided by the manufacturer of the hardware, and not the programming language, but still it is possible to create behaviour that resembles hardware interrupts and therefore its listed as one point. The second element that is extracted from the above sections is the need for control structures or easier ways to do decision making on background on the data that is obtained by the sensors. The third finding is the need for a system that can enable the sensors to push a message when they have registered a change or have crossed a threshold. The fourth finding is the need for a parallelism and or asynchrony, or at least the possibility to escape the sequentially. The

findings can be listed as this:

- Interrupts
- Control structures
- Push messages
- Parallelism

Part II

Introducing and Evaluating ThingML

Chapter 5

ThingML

After comparing different microcontroller languages and doing research on what the users finds important to have in a microcontroller programming language, it is time to find a language that is close to the demands that are found. One candidate to become a new leading microcontroller language is ThingML. This claim is based on the fact that ThingML meet some of the needs that have been found in chapter four, have several of the featured listed in chapter three and is still in development as an open source project. Since ThingML is open source, it is a good candidate to implement the suggested changes that arises from this thesis. ThingML stands for "Thing Modelling Language" and is developed at SINTEF and is being used in some projects at SINTEF, for example ENVIROFI, MODERATES, MoSIS and more[1]. ThingML is a language for the Internet of Things (IoT) *i.e.*, for embedded and distributed systems [44]. ThingML is based on a combination of architecture models and state machines to define the behaviour of components, and an imperative action language. One key principle of ThingML is that it should compile into running programs on several different platforms, *e.g.* Arduino, Scala, Java and C. While this principle is a well-established idea in model-driven engineering (MDE) and model driven architecture (MDA), current modelling tools usually fall back to plain text (not checked by the modelling tool) when it comes to describing the behaviour of states and transitions. This hand written code is usually directly written in the target language (C, Java) and often requires a advanced knowledge of the underlying framework for executing state machines *e.g.*, how to send a message in JavaFrame[2] or in Meta State Machine (C++ Framework)[3]. In ThingML is a complete action language and most of the behaviour is coded in a platform independent part. Note however that ThingML also provide a template mechanism to be able to mix ThingML code and target code (Java, C) and hooks to interact in both directions with legacy code written in the target language. This is particularly useful to define low-level components interacting directly with hardware, or high-level components interacting with a GUI (Graphical user

[1] http://thingml.org/pmwiki.php?n=Main.Research
[2] http://folk.uio.no/intime/ECSE2000JavaFrame.pdf
[3] http://www.boost.org/doc/libs/1_49_0/libs/msm/doc/HTML/index.html

interface *e.g.* in Java/Swing). One of the strongest arguments for choosing ThingML is that it does not limit the user to develop for one platform, like for example the *Arduino* development environment do to some extent. The *Arduino* development environment is designed for *Arduino* boards, or microcontrollers using ATmega chipsets[4]. TinyOS supports a wider range of microcontrollers but is still not as universal as ThingML aims to be. These restrictions bind the users for specific platforms and if the user wishes to develop for other platforms they often have to switch programming language and development environment. ThingML aims to provide the possibility to compile applications to different kinds of microcontrollers, smart phones and even desktop application. Currently ThingML supports Arduino, Posix c, Java, Scala and a few more languages, and provides seamless communication between those languages [19]. ThingML is also capable of producing adaptive firmware using a high-level adaptation DSL (domain-specific language) and aspect-oriented modelling techniques applied to state machines [18]. Other areas that were focused upon while developing ThingML were modularity, messages and event handling. For this language it was important that code written once, should be immediately available for reuse in other applications. Modules should be able to be linked together and reused. Event handling is also important and the ThingML language is based on events that trigger operations.

In this chapter the programming language ThingML will be presented, the rest of this chapter will be split in two main sections. The first section is about the platform independent part of ThingML, which makes up the runnable program. The second section is about the platform dependent part of ThingML, which binds the program to actual components on a specific platform.

5.1 Platform Independent Model

ThingML is created with simultaneous development for several platforms in mind, and therefore offers full platform independence through separation of the program itself and a platform configuration file. The platform configuration file maps the *things* and components used in the platform independent program to the platform specific properties that are available on the platform it is being developed for. This approach allows the programmers to develop one program that can run on different platforms. The need for a configuration file for each specific platform is still present.

In this section the components that are used in the platform independent model is laid out and presented.

5.1.1 Things

The main component in ThingML is the *thing* construct. The *thing* is mainly a software component but can represent a software wrapper of a hardware

[4]It is also possible to program ATTiny microcontrollers and a few other types with the Arduino language.

component, for example a light emitting diode, piezoelectric buzzer, an algorithm or a entire program. The things are completely modular and if for example a *thing LED* is created, this *LED* can be reused for all *LEDs* that are used in the application, circuit and even other applications. In other words, the generic behaviour of a *LED* is defined once and becomes available to be used in all applications. The internal behaviour of the *thing* is defined as a *state machine* within the *thing* component. A *thing* also have *ports*, these *ports* are used to send and receive *messages* to or from other *things*.

5.1.2 State Machines

A *state machines* is the main part of the *things* and can be seen as the equivalent to the *setup* and *loop* methods in the *Arduino* language. In ThingML the state machines does all the computing and work. The state machine can contain one or more *states*, *regions*, *component states* and *history states*. The state machine in ThingML is actually called a *"state chart"* and an initial *state* is required by the ThingML *state chart* as an entry point. In order for the state machine to either switch to another *state* or to re-enter the same *state*, a *message* must be received and matched to one *transition* defined within the *state*. A simple state machine that can toggle a light emitting diode on and off, can be seen in program 5.1, note that this example is not a complete ThingML implementation and will not run without additional code.

Program code 5.1 The state machine for a ThingML led blinker program.

```
statechart LedBlinker init Blink
{
    state Blink
    {
        on entry Timer!timer_start(500)

        transition -> Blink
        event Timer?timer_timeout
        action Led!led_toggle()
    }
}
```

5.1.3 States

States are the main building blocks in the state chart and ThingML provides three different types of *states* that can be used. They all have they different areas of expertise and can if combined create quite an intricate state machine.

States

A *state* is the simplest of the three provided state constructs in ThingML, it can contain four different constructs that can perform *actions*, and these are

on entry, *on exit*, actions during *transitions* and actions during *internal events*. The *on entry* construct have been used in program 5.1, in program 5.2 both *on entry* and *on exit* are shown. Both these constructs are optional in a *state*, and the *on entry* construct will be called each time the state machine enters the *state*, and the *on exit* construct will be called when an *event* causes the state machine to exit the state.

Program code 5.2 The on entry and exit points in a state

```
state ExampleState
{
    on entry do /*some action*/ end
    on exit do /*some action*/ end
}
```

Composite States

Composite states is a construct that allows one *state* to contain several *states* and thereby function as a state machine in itself. When the *composite state* is entered it goes into a internal initial *state* and will after that traverse it's *states* until a *message* is received that enables it to exit and return to a *state* in the main state machine. The *composite state* is defined with the keyword *composite* in front of the *state* keyword, and it must contain at least one *state*. An example of a *composite state* is shown in program 5.3

Program code 5.3 A ThingML composite state

```
composite state ExampleCompositeState init StateOne
{
   state StateOne {}
   state StateTwo {}
}
```

The composite state can also contain its own *on entry* and *on exit* actions that are executed before the initial state and before the exit of the composite state.

History States

A *history state* is an extension to the *composite state* with a memory of which *state* it was in the, last time the *history state* was exited. This mean that the *history state* have to define an initial state just as the *composite state*, this *state* will be entered the first time the *history state* is entered. The next time the *history state* is entered it will enter the *state* it was in the last time the it was exited. A example of a *history state* is shown in program 5.4. It is defines just like a *composite state* and the keywords *keeps history* is added to the end of the *composite state* declaration.

Program code 5.4 A ThingML history state

```
composite state ExampleCompositeState init StateOne keeps history
{
   state StateOne {}
   state StateTwo {}
}
```

5.1.4 Regions

A *regions* is a construct that appears at the same level as *states*. *Regions* are a construct that can contain several *states* just like the *state chart* construct or the *composite state*. The *region* will start to run its own state machine in parallel with the surrounding *state chart*. *States* inside a *region* cannot transit to *states* outside the *region*, but they can trigger *messages* that can be received outside the *region*. The *region* is defined by the keyword *region* followed by a name and then the keyword *init* and the name of a state defined inside the *region* as seen in program 5.5.

Program code 5.5 A ThingML region

```
region myRegion init StateOne
{
   state StateOne {}
   state StateTwo {}
}
```

5.1.5 Ports

The communication between *things* in ThingML happens through the use of *ports*. A port can both send and receive *messages* to and from other *things ports*. A *thing* can provide a port; this *port* then becomes available for other *things* to use. *Ports* can also be *required*, that means that a *thing* can state that it uses another *things port*. The communication through *ports* is done with *asynchronous message* passing that are presented in the next section. The *messages* that are sent through the *port* are the main way of triggering *transitions*, *internal events* and make the state machine change *states* and make the program go on. In program 5.6, an example of a *required port* is presented, this *port* is taken from the *blink* program provided in ThingML [17].

Program code 5.6 A required port in ThingML

```
required port HW
{
    sends led_toggle
    sends timer_start
    receives timer_timeout
}
```

5.1.6 Messages

The *messages* that are sent back and forth through the *ports* can be declared within a *fragment*, or directly within a *thing*, in order to be used by the *ports*. The declaration is the keyword *message* followed by the name of the *message*. The *messages* can also contain parameters, these parameters can be all of the data types that ThingML supports (see section 5.3.1). In program 5.7, an example of the *message* definition both with and without a parameter can be seen.

5.1.7 Interface / Fragments

If a *thing* has one or more ports that it *provides*, these *ports* can be defined as a *fragmented thing*, or defined in the *thing* itself. The name should ends with *"Msgs"* to signalize that this is the *interface*, see program 5.7. The *fragment* is created by the keywords *thing fragment* and are followed by an optional name. Inside the *fragment* all the *messages* that the *port* are supposed to handle must be defined.

Program code 5.7 The interface definition in ThingML.

```
thing fragment LedMsgs
{
   message led_on ();
   message led_off ();
   message led_toggle ();
   message led_brigthness (val : Int8);
}
```

This *fragment* with all the defined *messages* can now be used as an *interface* for the *things*, that means that several different *things* can include this *fragment*, use it as a *port* and send *messages* through it. At least one *thing* must implement the *port* as a *provided port* and another *things* may require it.

At this point everything that is needed to send a *message* is in place, a full implementation off how *messages* are sent and received through *ports* are shown in a larger example in figure 5.11.

5.1.8 Switching states

To make the state machine operational it must be capable of switching the current state, this is done by transiting when a specific message have been received. When an *event* has been triggered and a *message* has been sent through a *port*, there are two ways of catching and acting upon these *messages* in ThingML, it can be with either *transitions* or *internal events*. The use of *transitions* and *internal events* is the way a ThingML program advances and executes other *actions*. It is also possible to apply logical tests and *actions* to the *transitions* and *internal events* in order to make the state machines more sophisticated, these structures will also be presented in the following sections.

Transitions

Transitions are the main way of making the state machine transit from one *state* to another. The *transitions* maps a way from one *state* to another and must be defined inside the *state* that the *transition* are to be triggered from. When a *transition* is triggered the *states "on exit"* block will be executed if it is defined and then the *state* will transit from itself to the target *state*. If the *transition* has an *action* defined, that *action* will be executed after the *on exit* of the *state* and before the potential *on entry* block of the target *state*. A *transition* is defined by the keyword *"transition"* followed by an optional name and then an arrow followed by the name of the target *state*. The *transition* must to register for a specific *message* to listen to on a specific *port*, and the *transition* can not be triggered to execute before the *message* is received from this *port*. The *event* listener is declared by the keyword *event* followed by the port name, an exclamation mark and the message name. A complete example with both the definition rules and a transition definition can be seen in program 5.8

Program code 5.8 The transition definition in ThingML

```
//Rule
transition *optName -> targetState
event portName?message

//Example
transition -> stateTwo
event HW?timer_timeout
```

Internal Event

Internal events work almost the same way as *transitions*, the only difference is that an *internal event* does not transit the state machine from one *state* to another, it only catches the *event*, performs an *action* and resides back in the same *state*. In other words, an *internal event* does not exit and re-enter the *state*, this means that it does not trigger the *on exit* and *on entry* of the *state*. The only *action* executed by an *internal event* is the one defined within the *internal events* own *action* block. The *action* block will be explained in the next section.

Guards and Actions

ThingML allows several *transitions* and/or *internal events* to listen for the same *messages* within the same *state*, and therefore in order to get the right *transition* or *internal event* to trigger they can have a *Guard* element registered with it. This *guard* element is a logical test and if it evaluates to "true" the *transition* will trigger, if it validates to "false" the *message* will move on through the list of *transitions* that listen for the same *message* until it finds one that evaluates to "true", a *transition* without a guard element or if there is no more *transitions* or *internal events* left to check the *event message* gets discarded.

The *transitions* and *internal events* can also have their own *action* block. This *action* block is defined the same way that it is within the *on entry* scope of a *state* for example. If the *action* block belongs to a *transition* it will be executed after the *on exit* of the *state* and before the *on entry* of the target *state*. If the *action* block belongs to an *internal event* it is the only thing that is executed since the *internal event* never leave and re-enters the *state*. An example of *transitions* and *internal events* with both *guards* and *actions* can be seen in program 5.9

Program code 5.9 Transitions and Internal events with guards and action blocks

```
State exampleState {
    %omited code%
    transition -> State2
    event Port?message
    guard m.value == 1
    action do Port!sendMessage() end

    transition -> State3
    event Port?message
    guard m.value == 2
    action do Port!sendOtherMessage() end

internal event Port?message
guard m.value == 3
action do Port!sendThirdMessage(2) end
}
```

Functions

Sometimes the amount of code that goes into an *action* may become long or is duplicated several places; therefore it is possible to extract code into *functions* that can be called from *action* blocks. A *function* is placed outside the state chart and is defined with the keyword *function* and a name, the function can also take arguments. An example of a *function* is seen in program 5.10. *Functions* in ThingML cannot return values.

Program code 5.10 Function example with incoming variables

```
function toggleTwoLeds(lhs : Int16, rhs : Int16) do
    Led1!led_toggle()
    Led2!led_toggle()
end
```

5.1.9 Tying it Together

Now that the most important parts of the ThingML language have been presented it is time to show one functional ThingML application. For this purpose it is reasonable to use the *blink* program, and a *blink* program in ThingML would require three *things*, the first *thing* is a light emitting diode (Led), the second *thing* is the *blink* program itself. The third *thing* is a *timer*

element that can make the light emitting diode turn on and off at given intervals. In this example the assumption that a Led component and the timer component is available in the ThingML language is made. It is also assumed that this Led component provides a *port* that can receive *messages* that can make the light emitting diode to turn on, turn off and toggle on and off without our knowledge of which state the light emitting diode is in. A third assumption is that the *timer* provides a *port*. This port can receive a *message* that will set the amount of time and start the *timer*, and that it can provide a *message* to tell when the *timer* have reached it's target time. In program 5.11 a fully functional *blink* implementation in ThingML is shown.

Program code 5.11 ThingML blink implementation

```
import "../../hardware/bricks/led.thingml"

thing LedExample includes LedMsgs, TimerMsgs{
    required port Led{
     sends led_toggle
    }
    required port Timer{
     sends timer_set
     sends timer_start
     receives timer_timeout
    }
    statechart LedBlinker init blink{
        state blink{
        on entry Timer!timer_start(500)

            transition -> blink
            event Timer?timer_timeout
            action Led!led_toggle()
        }
    }
}
```

This program is only 19 lines long, and the state machine, which is the core program, is only eight lines long. This program is now compilable and ready to be deployed to any of the ThingML supported platforms, but in order for it to be possible a platform specific configuration file must be created and the components in program 5.11 must be mapped to corresponding components on the target platform. Another thing to notice here is that the LedExample *thing* includes *TimerMsgs* without importing it, it could have been handled in the platform specific file, but in this example it is not.

5.2 Platform Specific Model

The platform specific model is the part that binds the program to the platform it will run on and is called a *configuration* in ThingML. The *configuration* is used to map the *ports* in the program to *ports* within other *things* or hardware components. This file should be placed in a folder

within the folder that the platform independent program lies in. This folder should be named the same as the target platform with a underscore in front, for example *"_arduino"*. This file must contain several instructions in order to make a ThingML program compilable. The first statement that is needed, is to import the program, like the first line in program code 5.12. Then platform specific *things* that corresponds to the *ports* used in the program must be imported, this can be seen in line three and four in program code 5.12, here the timer is also imported. Then a *configuration* should be defined, this is similar to the main *thing* in the application, this is seen in program code 5.12 as the block of code named *configuration Blink*. Then the first thing that should be done is to make an instance of the application, this is done with the keyword *instance* followed by a name of the users choosing, and then a *colon* and the name of the *thing* it refers to. In this case, *LedExample* as shown in program code 5.11. The configuration can be seen on line seven in program code 5.12.

Program code 5.12 ThingML blink program platform configuration

```
import "../blink.thingml"

import "../../hardware/bricks/_arduino/led.thingml"
import "../../core/_arduino/timer.thingml"

configuration Blink{
    instance app : LedExample
    instance timer : TimerArduino
    connector app.Timer => timer.timer

    group led : LedArduino
    set led.io.digital_output.pin = DigitalPin:PIN_10
    connector app.Led => led.led.Led
}
```

Now it is time to tie the *ports* in the program to the *ports* provided by the Led *thing* and the Timer *thing* so that the program actually can interact with these. There are two different ways to do an *instantiation*, either a simple *instantiation* or a *group instantiation*. The simple one in this example is the *timer*, an instance of the *timer* is needed and it is created the same way as the *app* instance, see line eight in the program 5.12, the reason for the Arduino suffix is because the Timer *thing* in the file: *"/core/_arduino/timer.thingml"* is named *TimerArduino* and is an *Arduino* platform specific timer. Then it is time to define the connection between the *port* instances, this is done with the keyword *connector* followed by the port in the *app*, then a arrow (=>) and the port in the target *thing*. This is shown in line nine in the program 5.12.

Now it is time to do the *group instantiation*, the reason it is called a *group instantiation* is because many of the instantiations is already done in the *thing* that is used by the program and therefore it isn't necessary to do it again. The keyword *group* is used to state that it is a group instantiation, then comes a user decided name, a colon and the name of the *thing*, see line eleven in program code 5.12. The next thing to do is to set the *group* to a

desired IO pin for example. This is done by the keyword *set* followed by the group and an assignment of the pin in a ThingML enumerated type, see line twelve in the program. Then there is only one thing left to do and that is to connect the programs Led port to the port of the Led brick, this can be seen on line thirteen.

The configuration file is now complete and with the program file, it can compile to a running *Arduino* program that will run on an *Arduino* microcontroller. Now it is also suitable to cover the section that is listed as "*Language Construct, the Bare Minimum*" in chapter three. ThingML comes out with the metrics of two files, 30 lines of code and four statements.

5.3 Comparison and Evaluation

In order to make sure that ThingML is a programming language that is suited to be presented as a better choice for microcontroller programming than the three languages presented in chapter three. ThingML must be compared and evaluated on the same areas as the three other languages. In this section ThingML will be evaluated like the other languages, and in the next chapter ThingML will be compared to those languages and compared to the needs that were found in the interviews chapter four.

5.3.1 Variables and Type Safeness

ThingML provides by default most of the data types that are available in C, except *float*, the *Double* that ThingML provides only use 4 bytes and not 8 like a *Double* in C does, so it could be said to actually be a *Float*. Also, the use of the word *Double* for *float* data types is a design choice from the developer of ThingML. This was done to comply with the Arduino environment use of the *Double* keyword, which is a *Float* and not a real *Double*. The reason for ThingML to not compile to real *Doubles* in for example Java was to avoid the risk of errors during communications between ThingML programs that run on different platforms. ThingML also provides String, boolean and an enumerated data type called Digitalstate which can only be HIGH or LOW and therefore operates as a Boolean but are meant to map to electricity. Even the I/O pins available on the 20 pins *Arduino* boards have been enumerated so that the user can choose from a defined set of pins that exist on the *Arduino* microcontroller, this removes the possibility for selecting a I/O pin that does not exist. ThingML does not support dynamic typing, but it is possible to set an integer to equal "*hello*" or a string to be '*1*' without the ThingML compiler complaining. After ThingML have compiled however, the rules of the target platform and language apply.

When it comes to global variables ThingML does not offer true globalism, a variable is not available outside the scope it have been created in, and must be created within a *thing*. The variables are available in all underlying scopes, so a variable defined within a state chart is available to the underlying states and regions. In order to share a variable with another *thing* it must be sent through a *message*, and then the receiving end will only

gain a copy of the variable and not a *reference* to the same object.

5.3.2 Hardware Abstraction

The level of hardware abstraction in ThingML is rather strong and present, this is the case because ThingML is designed to compile to everything from desktop applications to smart phones and microcontrollers. Therefore ThingML must be suitable for all platforms and the hardware that might be accessed on one platform must be hidden from the other platforms. The hardware is only visible in the platform specific model when it is programmed for a platform where the hardware is essential.

5.3.3 Parallelism

ThingML does not support multithreading for microcontrollers, but it does when it compiles to languages that supports multithreading. Since ThingML is built strongly around events and have the possibility to have several regions in a state chart it provides a system that gives a feeling of parallelism even though nothing is ever executed at the same time.

5.3.4 Events

ThingML is built with *events* and *event* handling as a main aspect. Therefore all communication in ThingML happens through the triggering and receiving of events. In a ThingML program there is absolutely no way for a *thing* to communicate with another *thing* without triggering an event. The communication happens through asynchronous *messages* that may contain data values.

5.3.5 Modularity

Modularity was one of the main focus areas while developing ThingML and therefore ThingML is a completely modular programming language. One of the main features with ThingML is that if a software component is written once, it is available to be used in all following applications. This works especially well for hardware components such as light emitting diodes, rotary sensors, temperature sensors and so on. But it also works for example an algorithm, a login system or a special part of a program.

Chapter 6

Evaluating ThingML

In this chapter the ThingML programming language will be compared to *Arduino*, *nesC* and the *em* programming language on the six areas that have been outlined in chapter three. After this, ThingML will be evaluated to see if it meets the needs that were found in the user interviews in chapter four. After that my own findings from programming with ThingML will be laid out, before the opinions from two master-students that had to learn how to program with ThingML will be presented. In the end of the chapter, suggestions for improvements in ThingML from the data presented will be laid out.

It is a little strange to compare ThingML with the *Arduino* programming language since ThingML code can compile to *Arduino* code, but it is still a large difference in how ThingML code is written and what ThingML can provide, and that justifies the comparison.

The metrics that will be used to evaluate the ThingML programming language is listed here:

- Language Comparison
  - The bare minimum
  - Variables and type safeness
  - Hardware Abstraction
  - Parallelism
  - Events
  - Modularity
- User Interview results
  - Interrupts
  - Control structures
  - Push messages
  - Parallelism
- My findings
- Master-students experiences

6.1 Evaluation

6.1.1 Comparison with Arduino, nesC and em

In this part ThingML will mainly be evaluated against the three other programming languages to find out if it is stronger or weaker in the six areas. The first area to be evaluated is the language construct and the minimum amount of code needed to write the *blink* program. Here the line count and the number of executed lines will be compared with the blink examples written in the four languages.

Table 6.1: Blink example comparison

	Arduino	em	nesC	ThingML
Lines total	9	23	29	30
Statements	5	8	10	4
Number of Files	1	1	2	2

Length and Complexity

As seen in table 6.1 ThingML and *nesC* uses two files to compile one program, but in ThingML this is actually not necessary as both the platform independent and the platform specific part can be written in the same platform specific file. Then again, the possibility to compile the same program to several different platforms is lost. In the scope of this thesis that only concern about microcontrollers, it is enough to write everything in the same platform specific file, and ThingML will be evaluated from the point of just needing one file.

When it comes to number of lines of code, it is clear that again it is *nesC* and ThingML that is the worst ones, and *Arduino* have a clear lead. With regards to the number of statements ThingML is in the lead with only four lines. This means that ThingML have a large amount of overhead in the amount of lines of code needed to get something to work, but this overhead will not grow very much if the program is extended, but it will grow. ThingML also have a feature that is also found in *em*, namely that it is not needed to use semicolons at the end of lines (with one exception in ThingML). In total Arduino is the winner of this part, and second place is going to ThingML. One could argue that the second place could have gone to em, but the amount of executed lines is so much larger then in ThingML and is bound to grow together with the overhead which will make larger programs larger then ThingML programs, therefore ThingML is slightly ahead of *em*.

Variables and Type Safeness

The second point of this evaluation is regarding the kind of variables available and the type safeness of these. All of the four languages provide the same data types that are found in C except ThingML and *Arduino*,

which currently does not support real *doubles* that are eight bytes long. In ThingML this is a design choice made so that the code conform to the *Arduino* code, and to make communication between *things* programmed for different platforms will go as expected. Only Arduino and ThingML provides the String data type. In all four languages users can write custom modules/structures or objects that can be used as a data type. In ThingML it is easy to create custom data types, it is done directly within a file called "*datatypes.thingml*" and those data types will instantly be available to use. In that file it is also possible define what native data type it will correspond to in the different languages ThingML can compile to. This point goes to ThingML for providing a great way to create custom data types and enumerations. When it comes to global variables only *Arduino* and *nesC* provides the use of these, but whether or not that is positive is difficult to say. For example in *nesC* which is designed to run with several threads at once there could occur problems if a variable was accessed and/or modified by different threads at the same time. In Arduino, which always run sequentially the same problems will not occur, if variables are reused by accident, this can cause problems. Then again by using global variables there might not be a need to send the same variable between several methods and the user can end up writing less code. So the use of global variables could be a drawback and something useful therefore it is a tie between the languages that provides global variables and those who don't.

The last part regarding variables, are how type safe they are. All of the languages provide C constraints, which means that it is possible to set an integer to take a long value and risk a data loss. One interesting thing with ThingML is that it is possible to set an integer to be "*hello*" and the compiler or editor will not say anything about it. This is because the program might compile to a language that don't have this constraint such as JavaScript[1]. Since ThingML generates source code that must be compiled by the native languages compiler, this check is left to the native language compiler instead and it will react to the assignment if necessary (and the constraint is implemented). Regarding variables and type safeness ThingML earn a small point for providing an easy way to define own data types, and for having a check on the I/O pins used. The rest of the areas covered in this section is a tie between the languages.

Hardware Abstraction

Next up is hardware abstraction and in table 6.2 it can be seen that all four languages provide hardware abstraction. In the comparison part in chapter three it is stated that *Arduino* take the hardware abstraction the furthest, but when ThingML is in the equation, ThingML takes a clear lead here as the language can abstract the hardware away entirely in the platform independent part. This is a direct result of the fact that ThingML can compile to different platforms and thus can run on a regular computer

[1] even though JavaScript is not supported at writing time

without GPIO pins. In the platform dependent part of the language there are clear references to hardware, as pin mapping has to be done manually since it would be unwise to randomly assign GPIO pins. ThingML does however not directly give the users that high level programming language feeling, but it can be several reasons for that, for example the simple design of the editor (figure 6.1), or the lack of ”normal” high level language constructs such as the use of the keyword *new*. In table 6.2 it is shown that Arduino is the better language here as it also gives the high level language feeling.

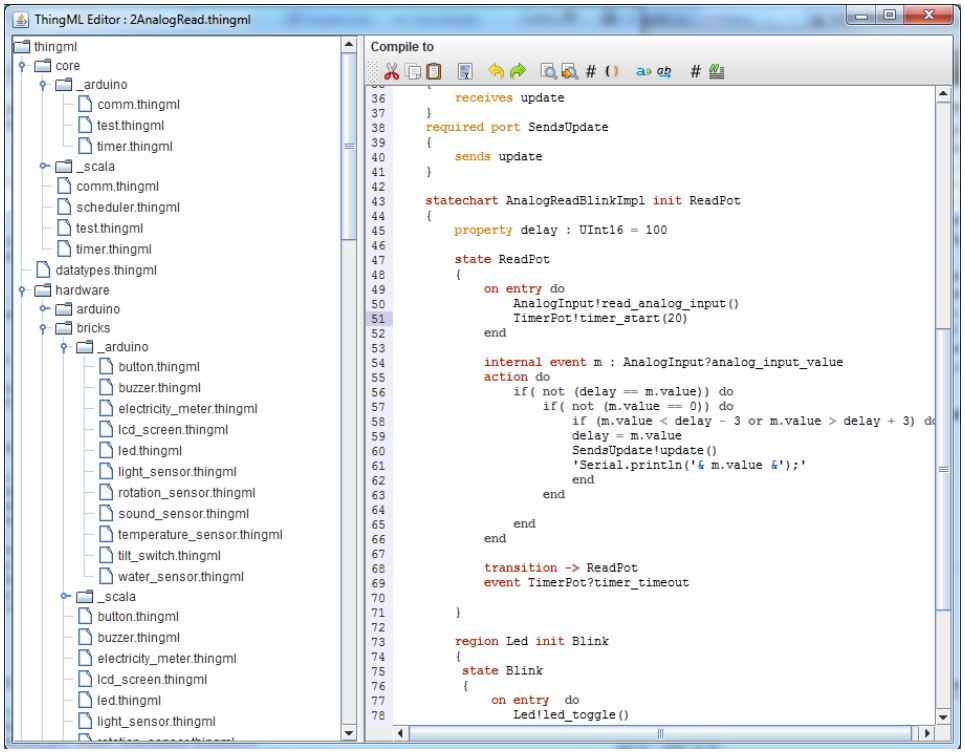


Figure 6.1: The ThingML editor

Table 6.2: Hardware abstraction comparison

	Arduino	em	nesC	ThingML
Hardware abstraction	x	x	x	x
Low level programming	x	0	x	x
High level feeling	x	0	0	0

Parallelism

Parallelism is an uncommon thing to find in the world of programming languages for resource scarce microcontrollers, but still it exists. *nesC* does provide multithreading and is also designed specially for the purpose of multithreading. The three other languages however, does not provide this.

There exist third party libraries for *Arduino* that can enable multithreading capabilities, but not true parallelism. ThingML and *em* does not provide any multithreading, but still, the way that these languages are built based on *asynchronous messages* and *events* with a underlying controller that handles the program. It is possible to write parts of the program as if they are executing in a threaded mode, but still no real parallelism exist. Table 6.3 contains a ranking of the languages based on multithreading support. In this table ThingML and *em* are ranked above *Arduino* since it is possible to use threading logic while programming as code in the different modules/things don't need to call each other to execute.

Table 6.3: Multithreading support comparison

	Multithreading Support
nesC	Natively
ThingML	Not at the current time, but behaves like it.
em	Not at the current time, but behaves like it.
Arduino	Third party library

Interrupts

All of the four languages support hardware interrupts and will jump from the current code execution and to interrupt code if an interrupt is triggered from hardware. When it comes to software events there is one language that lags behind, namely the *Arduino* language. *em* comes before *Arduino* with support for using events through the `EventDispatcher`. On top, are ThingML and *nesC* that are built around the use of events, and uses events as a main mode of communication. ThingML is build more strongly around events than *nesC*. *nesC* can also make use of methods to advance in the program flow which ThingML can't since events is required to allow the program flow to progress. With respect to interrupts ThingML is slightly better than *nesC* with *em* right behind and *Arduino* comes last.

Modularity

Modularity is the last of these six comparison points. As stated in chapter three Arduino does not offer modularity in any extent besides the libraries that are provided in the bundle, and copying and pasting is the main way of reusing code. *em*, *nesC* and ThingML on the other hand are languages that are build around the concept of modularity and the reuse of code and modules is a main point in these languages. This can be seen quite clearly in table 6.4.

Table 6.4: Modularity support comparison

	Arduino	em	nesC	ThingML
Supports modularity	No	Yes	Yes	Yes
Facilitate reuse of code	No	Yes	Yes	Yes
Module based	No	Yes	Yes	Yes

6.1.2 Meeting the Users Needs

Chapter four presented the results of interviews with users who had worked on projects using an *Arduino* microcontroller. From these interviews four needs were extracted. The first need extracted from the interviews is interrupts; this is intertwined with other of the needs as well, as stated in chapter four. Real hardware interrupts is hard to do something about in software and the type of programming language used have no effect on the number of hardware interrupts available. The way ThingML is built can weight up a little for this and meet the users need to some extent. Because the programs are built with *things* that are connected together in various ways and every *thing* have it's own execution cycles, one thing can "constantly" listen to a specific port and trigger a *message* at the moment it receives input from the hardware. Because of the scheduling, this is not a method that guaranteed will pick up a hardware event. If it happens too fast it is not a safe method to use instead of real interrupts. If the hardware event is long enough and the program don't have to many *things* that require processor time it should work to some extent as a substitute for the lack of hardware interrupts.

The second need that was extracted from the interviews was the need for a better control structure to easier make decisions based on the sensor inputs. This is a tough nut to crack and probably has more to do with the student's ability to write good checks than the language itself. Since this was a direct issue in one project and a partial issue in another it should be evaluated as well. ThingML does not directly provide any better control structures, but because of the way that ThingML programs are built up with *things* that can consist of direct sensor communication or of several other *things*, it is possible to build hierarchies of *things* that can combine sensor readings and create better evaluations. Still, this probably have more to do with the programmers experience and ability to create these structures.

Push messages from sensors is something several of the students wants to have. This is a key factor in ThingML and is better known as *events*. *Things* can be set up to send a message through an event when it reads a specific value or a specific event occur, this behaviour is exactly what the student said they wanted to have in a programming language for microcontrollers. So the use of ThingML for this purpose would be just right.

Parallelism is the last of the four things that were extracted out of the interviews. ThingML does not provide any parallelism, but can at least provide a behaviour much like multithreading, and can emulate

parallelism. The *region* construct is great for this purpose.

6.1.3 My Findings

In order to fully be capable to evaluate the ThingML programming language it was important for me to learn how to program with it as well. To start out, I made small programs that communicated with different sensors and created output, after that I went on with slightly more complex programs that used concepts such as *regions* and *composite states* and tied more and more sensor and output devices together in the same program. After a while I felt comfortable with most of the elements and structures while programming in ThingML, but there is one aspect of the language that I struggled a lot with, and that is the platform dependent configuration files. I found the concept itself to be quite easy, but I still found it quite hard to follow the correct connection paths all the way to the right *ports* in the *things*. To give an example of this, I wrote a program that makes use of a button. In order to map that button I had to write this:

```
group button : ButtonArduino
    set button.io.digital_input.pin = DigitalPin:PIN_3
    set button.io.digital_input.pullup = true
connector app.Button => button.button.Button
```

The first three lines are quite easy, but the fourth line has caused lot of frustration. To break the line down, the first part states that my `Button` port of my program, referenced to as "`app`", is to be connected to `button.button.Button`. To figure out that `button.button.Button` is the right connection, I did the following: First, what the group definition is, must be found, and that is *"button"*, as seen in the first line. Then a look must be taken at the button definition found by the import statement. In the file located at the end of this path: "`thingml/hardware/bricks/_arduino/button.thingml`" these two lines of code are found: "`import "../../io/_arduino/digital_input.thingml"`" and `instance button : Button`. The last line says that the second part of the connection should be "button" and the last element of the connection statement can be located in the file referred to in the first line. By looking in that file, a port named Button is found, as stated in the button configuration file. Still it was necessary to look in that file in order to make sure it was not referring to a port or connection located elsewhere. This process of finding the right connections can be confusing and time consuming.

While I was in the initial learning phase I tried to make a small program that would take input from a potentiometer and make a light emitting diode blink with a rate equivalent to the value that was read in from the potentiometer. This proved to be a little harder than what the same program would have been if written in the Arduino language. In the Arduino language that program could have been written in just fourteen lines, and is available in appendix G. After figuring out that a region was needed in order to make this work in a efficient way, it was implemented, and then I found out that this actually opened for a new and improved

behaviour in the circuit. Namely that if the potentiometer was turned during a time when the leds state was supposed to be static, it became possible to cancel it and directly change the blink interval with immediate effect. This gave a more responsive behaviour. Thus the program were a good part longer with over 60 lines in the platform independent part and 20 lines in the platform dependent configuration file. Most of the lines in the platform independent file is however port and message assignments that are there to enable communication between *things*. The state machine itself is not very large. This improved behaviour was a great advantage for me and opened up a whole new way to think about microcontroller programming. This way of thinking and writing the programs could have changed several of my earlier programs and circuits behaviour quite a lot.

ThingML is a young language and many elements that in my opinion should have been implemented, is not. A good example of this is the fact that a check to see if a value is less or equal to another value, or if a value is greater or equal is not implemented. Instead the user have to first check if the value is less and then if it is equal. This often causes the *if* test to become twice the length that it has to be. To implement this should be a quick task for the developers and from my analysis it would be worth to consider.

When I started to program robots with ThingML I came across a negative element in the language. This was the duplication of transitions that were valid in several states, they were cluttering up my programs. This can occur for example in a robot that can perform several different *actions*. Each *action* has it's own *state* that handles the execution of that *action*, and the robot is supposed to change to a specific behaviour if it receives a specific event from a sensor regardless of which *state* the program currently resides in. Then the same *transition* have to be defined within every single *state*, and this means that it can be a lot of duplicated code which will become an obstacle if the code needs to be altered or debugged.

This problem with replicated transition statements is also something that the creators of ThingML had stumbled upon and there is a comment in one source file from the early version of ThingML that says:

//This block is in all states, we should be able to write it only once [16]

This comment is actually found four times in that file since it have been copied along with the block containing eight lines of code that is duplicated three times and equals 32 lines of code.

In order to make fully use of the equipment and hardware I have and want to use with ThingML, I had to learn to convert Arduino libraries into ThingML *things* that I later could use as modules and interfaces. Two hardware components I worked with was the SOMO14D (seen in figure 6.2) and the ultrasonic sensor (seen in figure 6.3). The SOMO14D module is a device that can play sounds stored on a memory card; it is controlled by simple TTL operations. The ultrasonic sensor is used to measure distance to objects by the use of sound, and it can measure up to 2500 centimetes. After implementing these two components into ThingML *things* I was left with two quite different experiences, and to start of with the positive one; Apart

from some struggle with the platform specific file, the implementation of the SOMO14D went easy. The sound chip was easy to interface and to control, in fact it became easier to use with the ThingML language then with the Arduino programming language because of the way ThingML uses *messages* to communicate between *things*.

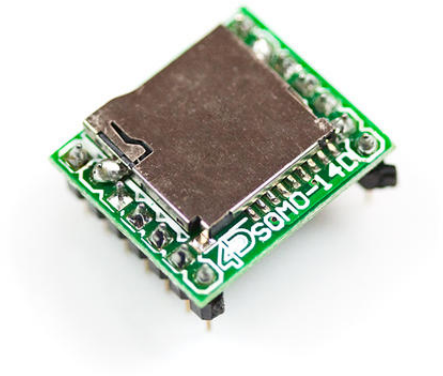


Figure 6.2: SOMO14D module

The ultrasonic sensor on the other hand was a quite different case when it came to the implementation. The platform specific configuration file was written without severe issues, and the writing of the *thing* went on satisfactory, but because of how the sensor works, I had to inject a lot of Arduino code in the *thing* component in order to make it work. The ultrasonic sensor works by sending a short burst of sound and then waiting for the sound to hit something and bounce back and hit the sensor. Then the distance can be calculated by the amount of time it took from the sound was sent out and until it came back. Since ThingML is supposed to be non-blocking, Arduino code that could listen on a pin until change had to be injected along with code that managed waiting. A part of this code can be seen in program code 6.1

Figure 6.3: The Ultrasonic range sensor

Program code 6.1 Snippet of the Ultrasonic sensor implementation

```
thing fragment RangeMsgs {
      message get_range();
      message range(cm : Int16);
}

thing Range includes RangeMsgs, DigitalOutputMsgs{
property duration : Int16 = 5
property distancne_cm : Int16 = 0

    provided port RangePort {
        receives get_range
        sends range
    }
    required port DigitalOutput
    {
        sends set_digital_output
    }

    statechart RangeImpl init Sense
    {
        state Sense
        {

         transition -> Sense
         event m : RangePort?get_range
         action do
            DigitalOutput!set_digital_output (DigitalState:LOW)
            'delayMicroseconds(2);'
            DigitalOutput!set_digital_output (DigitalState:HIGH)
            'delayMicroseconds(10);'
            DigitalOutput!set_digital_output (DigitalState:LOW)
            duration = 'pulseIn(12, HIGH, 100*29*2);'

            distancne_cm = duration / 29 / 2
                'delay(100);'
             RangePort!range(distancne_cm)
         end
        }
    }
}
```

Another discovery I made while doing some testing with the ultrasonic sensor was that I could not connect more then three of them with the implementation that I had written. This occurred because of the amount of messages the sensors ended up sending to the main *thing*. The program made the *message* queue overflow and *messages* became lost. This behaviour shows a limitation in ThingML on handling large amounts of messages during a short timespan. It is however possible to configure the *message* queue to hold more *messages*. The drawback might be that the application stops working in real time and the application will seem to lag behind.

While programming a robot I stumbled upon what I think is a flaw in ThingML, this has more to do with the editor and the compiler than with the programming language itself. A sensor I had connected seemed to work as it should. The values that the sensor read in was clearly displayed in the console, but the robot did not respond to the values at all. What I then found out was that it is allowed to instantiate several instances of a ThingML program in the platform specific file. What I had done was to

write two lines different places in the configuration file, the lines were:

```
instance app : BigRobot
instance robot : BigRobot
```

Then I connected all sensors and motors to the "app", and one sensor to "robot" which both points to the same ThingML program. Because of the different names, it is two instances of the same program, which therefore can not communicate without mapping ports between them. The scheduler ran both program instances and wrote out debug messages that made perfect sense, but the actual behaviour of the robot made no sense.

6.1.4 Experiences from Master-Students

In order to get a wider understanding of what that is positive and negative with the ThingML programming language, two master-students learned how to program with ThingML. One of the students had a wide experience with microcontroller programming. The other student did not have any previous experience with microcontrollers. Both had good knowledge of high and low level programming languages such as Java, C and Assembly.

The key concepts; *state charts*, *things*, *ports* and *messages* were explained to them and they started out programming some simple circuits such as blinking LEDs, dimming LEDs with light sensors and so on. They had some knowledge of how state machines work, so that concept was not entirely new to them, and they were told that ideally one state is only responsible for one action, for example in a robot there would be one *state* for driving forward and one *state* for driving backwards. Another example is LEDs were one *state* should turn the LED on, and one *state* turn it off.

The part of ThingML that they struggled the most with was the language dependent configuration file and they often needed help to get the connections right. They understood the concept, but were not able to to learn it and use it correctly without extensive guidance.

The two students did complain some about not being able to use "<=" and ">=" in their *if* tests and either had to write two tests or combine to tests in one statement. Another aspect of confusion and irritation was that they had to write "do" and "end" around action blocks, but use curly braces in the scope of *things*, *state charts* and *states*. The students said that it was strange to write "do" and "end" and it was more time consuming to type five characters rather than two curly braces.

From my experience, students often can be prone to do as little work as possible and some times they end up spending more time on finding a way to do as little as possible than actually doing their work. This was exactly what happened when they started to realise that in order to have the possibility to reach one *state* from several other *states* they had to duplicate the same *transition* several places, they became creative and instead created *states* that could do several *actions* instead. This approach also added some complexity and need for variable checking.

In regards of consistency there are some mismatches in ThingML. As stated by the students, there is a miss match in the way the sending and

receiving of messages are written. For example when register to receive a message this is written:

```
MyPort?incomming
```

while sending a message is written like this:

```
MyPort!outgoing()
```

There are two differences here, one is the usage of ”?” and ”!” which symbolise if the port is sending or receiving messages, the other difference, which the two master-students did not like, was that while sending messages writing parentheses is required, but while receiving messages parentheses is not required. The students meant that this is inconsistent and that the parentheses should always be written regardless of if it was a receiving message or a sending message. They also mentioned that semicolons were used inconsistently, which must be written after the *message* declaration, but no other places in the platform independent model. The suggestion here is that the inconsistency should be removed to keep the language consistent. The last error source they found in the language was that if a *port* that should be able to send or receive a *message* is defined, but this *message* is never used in the programs state machine, and the program will not be able to compile because of lacking dependencies. ThingML will generate code, but the platform specific compiler will complain.

6.2 Evaluating Results so far

In this section the results that have emerged from the language comparison, how ThingML comply with the users needs, what I have found during my usage of the ThingML programming language and the master-students points of view, will be presented.

6.2.1 Results From the Comparison

After evaluating ThingML with the other three languages the results show that ThingML is not significantly better than the other three languages, but it did not come out worse either. This comparison was based on aspects of modularity, multithreading, events, hardware abstraction, variables and type safeness and the amount of code required to make a LED blink. Based on the results, ThingML is the language that offers the most complete package compared to the other three languages. The area where ThingML came out as the worst candidate was with the basic blink example where ThingML had 30 lines of code compared to Arduino which had only 9 lines. However the amount of statements needed to accomplish the same behaviour was one less with ThingML than with Arduino (ref table 6.1) which shows that a lot of the lines in a ThingML program is not due to the logic and working of the program, which usually is the hard part to get right while programming. Based on these results ThingML can be a candidate to become a leading microcontroller programming language.

6.2.2 Results From the Users Requirements

The language requirements that were extracted from interviews with students that had or were working on projects were compared to what ThingML can provide of functionality. The results that came out of that comparison shows that ThingML could have made their programming easier based on ThingML's event functionality and asynchronous behaviour as this could have made it easier for the students to obtain the behaviour they wanted. This shows that ThingML can provide functionality that can speed up development time and enable users to write more sophisticated programs to cope with their needs. There is however no clear indications that show that ThingML provides better control structures than the other languages.

6.2.3 Results From Programming

The results from my findings and the feedback from the two master-students who learned to program with ThingML will be combined in this part since many of the findings are the same and the results have emerged from the same approach.

Since ThingML is a rather new and young language, with currently ongoing development, it still have some bugs, unfinished parts and design choices that can be subject to change if the developers decides to do so. It is therefore important to point to issues and problems that occurs to make the developer aware of them and subject to change.

To start of with the part of ThingML that caused the most irritation and problems, the language specific configuration file is hard to work with. It takes a long time to fully learn how to connect different *ports* together and often the errors the users do, don't show up before runtime, and then nothing happens. Some times the editor will tell the user if what he or she is doing is fundamentally wrong, but it will not provide feedback if what the user is doing is slightly wrong. Luckily the developers are also aware of how difficult and confusing the language specific file can be to write, so a process of implementing a graphical user interface with dragging and dropping of connection between ports have started [27]. To achieve this, a framework from the *Kevoree* [21] project is used. *Kevoree* is a project that aims at enabling distributed reconfigurable software development and is doing this with a graphical user interfaces that suits ThingML very well.

When it comes to smaller aspects of the language, such as writing parentheses on messages that are being sent and not on messages that are being received, and having to write two asserts in an *if* test in order to check if a value is equal or higher or lower than another value, the developers are postponing the implementation of these things. Currently ThingML is being developed in close connection to other projects that the developers are working on. This has for now resulted in focus on how to make the language benefit these projects, and thereby a focus on bigger concepts rather than the combination of two logical checks.

The next issue is the duplication of transition code, the language

currently demand the same code to be written and duplicated if one *state* is supposed to be reachable from several other *states*. This is something that me, the two other master-students and the developers of ThingML do not like and that makes the program code larger than necessary, harder to maintain and make the whole state machine more complex. Therefore this is a topic that should warrant further research.

6.3 Improvement suggestion

The most prominent part of ThingML that requires improvement is, from the results in this thesis, the way transitions is handled. ThingML programs that have several *states* with the same *transition* declaration that lead to the same target *state* can benefit from this. In order to fully understand the problem it will be useful to look into some examples of how the current way of writing transitions can cause problems. First two guidelines for programming state machines is declared;

- When programming state machines, they should be programmed so that one *state* is responsible for one set of coherent *actions* only.
- That all *states* should be able to transit directly to another *state* that contains an *action* set which can execute after the current *action*.

These guidelines are made so that the state machine can be easy to understand and easy to rewrite in terms of behaviour. But by following this guideline a simple state machine can become quite complex with respect to *transitions*.

To start of with an example, lets imagine a robot, a robot that can walk, run, stop, talk, turn on and turn off. In order to create this robot with a state machine that supports the guidelines made above, it would quickly become complex even though it isn't really that big compared to many industrial systems. A state diagram of a robot with this behaviour can be seen in Figure 6.4. This robot can perform any action in any sequence, resulting in a highly connected graph. The robot has six states and twenty transition, and it can be read from the graph that many of these transitions are duplicates. Even though the number of states in this state machine is low and the behaviour is simple, the complexity caused from the significant amount of duplicated transitions can make the state machine difficult to maintain.

In this particular example, since the current state (except Idle) of the robot does not affect the states it can reach (basically, all others, except turnOn), Event-Condition-Action (ECA) [6, 11] rules are more appropriate. ECA rules would make it possible to factorize most transitions: t_turnOff, t_run, t_walk, t_stop, t_talk; t_turnOn being a "classic" transition only appearing between idle and turnOn. This combination of ECA rules and "classic" transitions can be modelled in a state machine in two different ways.

The first way uses a master state that delegates all transitions out to other *states*, and the *states* return back to the *master state* upon completion

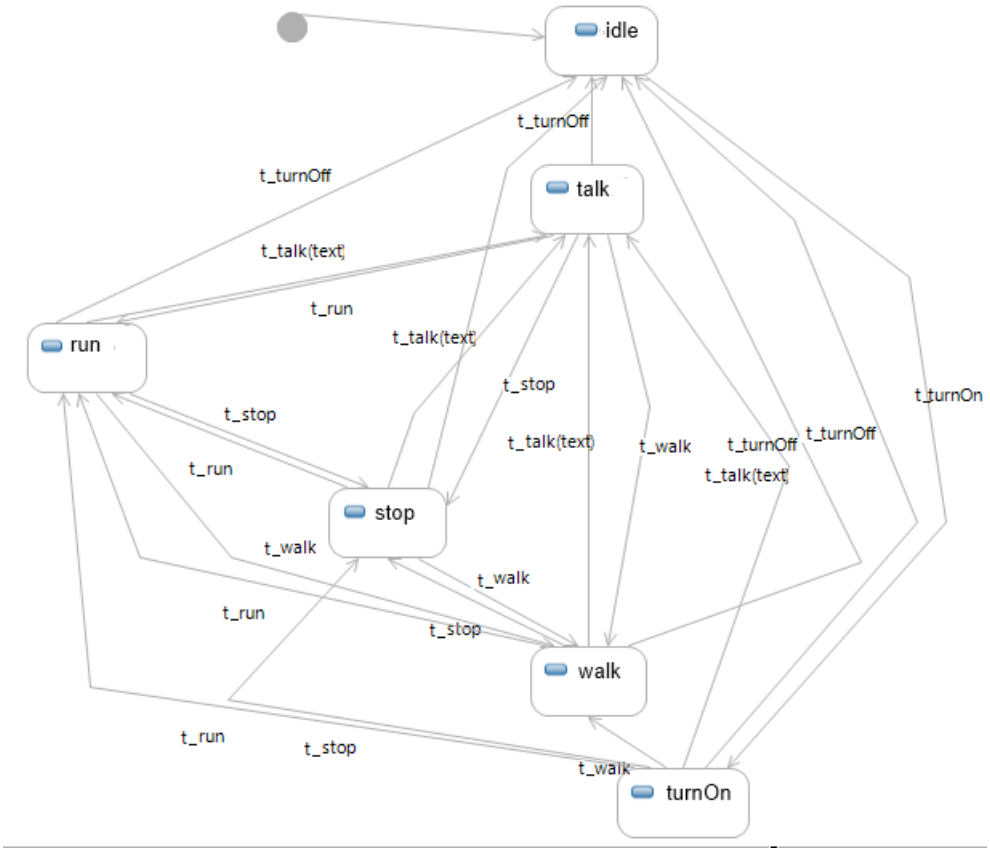


Figure 6.4: A simple robot

of their assigned *actions*, see Figure 6.5 [38]. This results in one large *state* in the code that handles all *transitions*. The approach forces the state machine to do *transitions* back to the *master state* that in reality should not be necessary as the state machine should be able to move directly from the *walk state* to the *run state*, for example. These back and fort transitions basically correspond to ECA rules.

The second common way of programming state machines consist of collecting all the actions into one super state that contains all the logic of the system, as well as a large amount of self transitions, see Figure 6.6 [38], where each transition basically correspond to one ECA rule. This approach, however, present some problems. Firstly some states might go missing; in the example shown in Figure 6.6 the stop state have disappeared. It is possible to argue that in order to turn off, the robot must stop, but in this example it means that the robot can't stop without turning off, which again means that the robot cannot stand still and talk. Arguably it is easy to just put another self-transition in the state machine in order to get the "turn off" action, but this is a real example from a state machine tutorial [38] and will therefore remain unchanged. The other problem is that the "Activity" state itself has no actions, the actions are put in the transitions instead, and the state itself does nothing but trigger transitions based on its input.

The two state machines, Figure 6.5 and Figure 6.6, illustrates one major problem in a full-ECA approach, also reported in Fleurey and Solbergs work [15]: as the system is getting more complex (for example with more ECA rules), it is becoming very difficult to understand or validate the system as it is difficult to tell which state the system currently resides in. It is of course possible to infer and generate the state machine [15] corresponding to a set of ECA rules (for example a state machine equivalent to the one presented in figure 6.4, but then it becomes difficult for human beings to understand the state machine because of the large number of cross-cutting transitions. *Cross-cutting transitions are to be defined as: a set of transitions* t_i *(i=0..n,* $n \geq 1$*), having the same target state TS, the same trigger T, the same guard G, and the same behaviour B (same sequence of actions to be executed when the transition is fired).*

To see how the use of cross-cutting transitions can clutter up a system it is useful to look at another hypothetical example. Take a system were every state must be able to transit directly to another state. With a small state machine with two states there is only two transitions, in a slightly larger state machine with four states, the number of transitions would grow to twelve. This is a significant increase, and of those twelve transitions there are only four unique cross-cutting transitions. This might not sound like much, but when the state machine is extended to contain eight states, which in itself aren't really that many in terms of state machines, then it would be 56 transitions. With such a large grow in the amount of transitions the state machine quickly become incomprehensible for humans. If one transition have to be changed, the change is done inside each state. This means that in the example with eight states, the same change needs to be done eight times instead of just one. Figure 6.7 and 6.8 illustrates this, when there are two states and two transitions the diagram is not very complicated. When

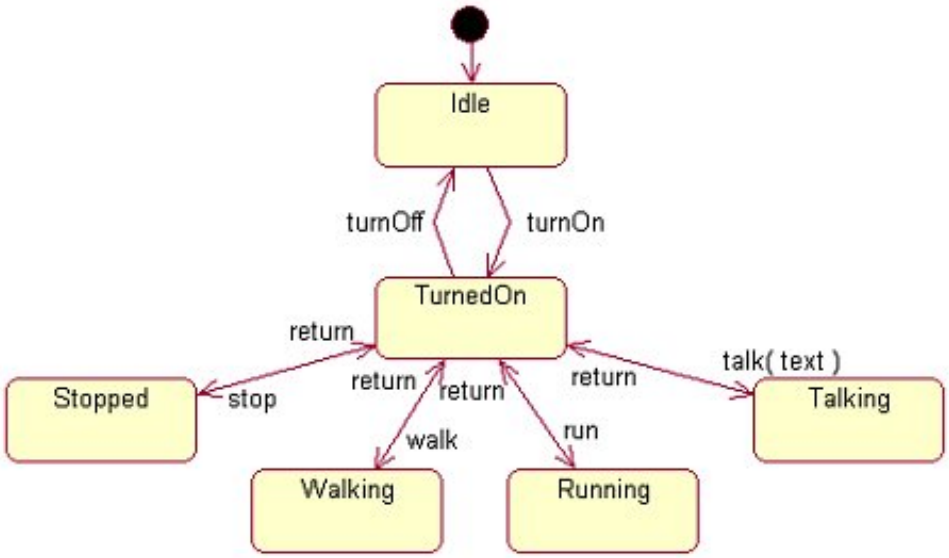


Figure 6.5: A simple robot with a master state.

there are eight states and 56 transitions it is virtually impossible to take a look and see if one transition is missing. Even though this graph is made symmetric it is still hard to see and fully comprehend what is going on, and if the model had been more complex it would have been even harder to fully understand the state machines behaviour. Also it is instructive to imagine how the code would look, given the assumption that one transition consist of two lines of code, which is the minimum in ThingML, each state would contain 14 lines of transition code, and adds up to 112 lines of code in all eight states. With such a huge amount of duplicated code, further development and maintenance of the code at a later stage might be difficult and cause problems. In these examples, self-transitions are omitted, but if present, they would help making the state machine more complex. If the code is inconsistent written, as it often might be with little experienced programmers, it may become even harder to comprehend the code and the state machine.

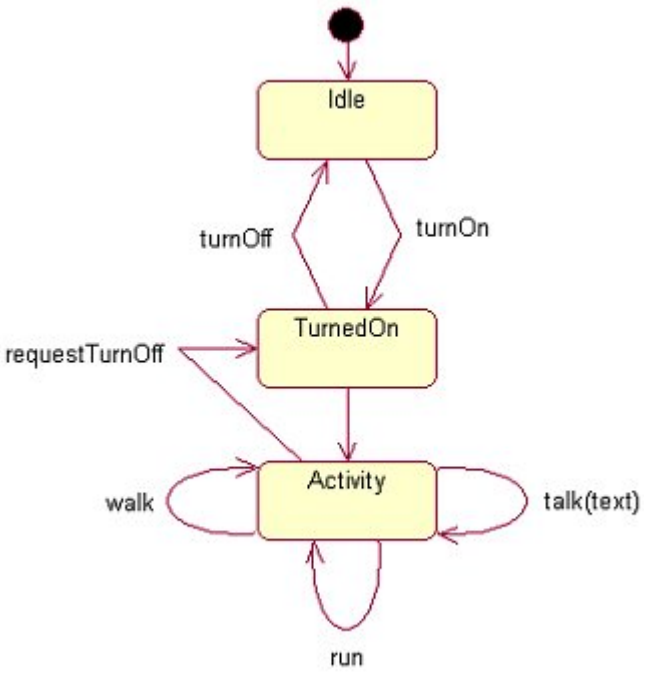


Figure 6.6: A simple robot with one action state.

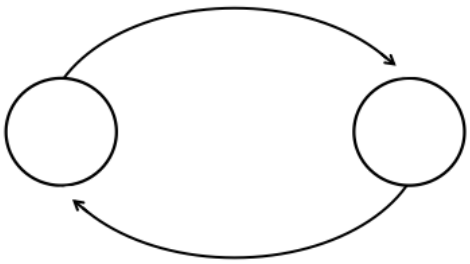


Figure 6.7: A state machine with two states and two transitions

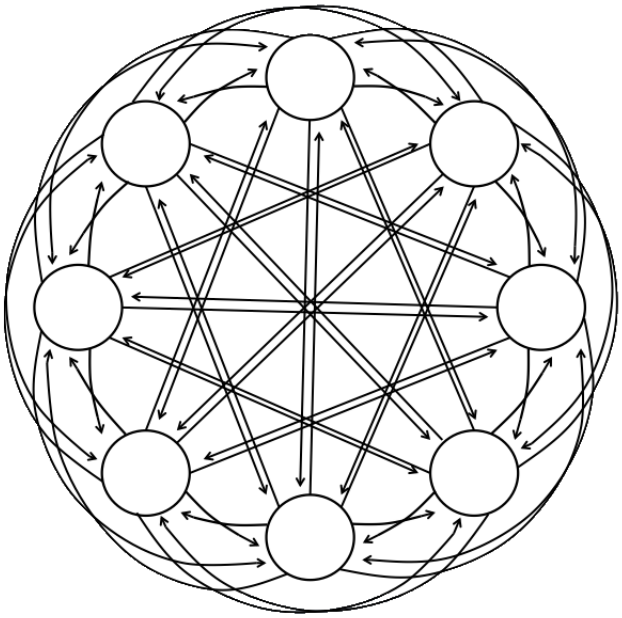


Figure 6.8: A state machine with eight states and 56 transitions

Part III

Introducing Factorized Cross-Cutting Transitions in ThingML

Chapter 7

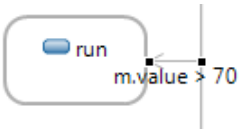
ThingML Improvements

As seen in the previous chapter, cross-cutting transitions can clutter up a state machine to such a degree that it can become incomprehensible for humans, and it can become very difficult to maintain a state machine and make changes to it. In this chapter, one possible way of implementing factorized cross-cutting transitions in ThingML is presented. Since cross-cutting transitions is something that occurs in all state machine languages, this approach could be implemented in any language and therefore there will also be examples on how factorized cross-cutting transitions could be implemented in the unified modeling language (UML) [25], also the same examples shown in UML state machine diagrams can make it easier to fully understand the approach.

7.1 Factorizing Cross-cutting Transitions

The point of factorized cross-cutting transitions is to gather *transitions* which share a common target *state*, *event*, *guard* and *action*, but has different source *states*, into one scope. The idea is to define such transitions within the container of the relevant *states*; either the *state chart*, a *composite state* or a *region*. The semantics for a factorized cross-cutting transition is equivalent to distributing the transition to all *states* directly contained in the *state chart*, *composite state* or *region*. The transition is not distributed to states within sub-states or sub-regions. An example of the graphical and textual syntax for the declaration of factorized cross-cutting transitions is available in Figure 7.1 both in UML and ThingML code. The example shows how the transitions responsible for sending the robot to the run state is extracted and put into a global declaration. In UML it would look like the transition arrow is coming from the surrounding element. In order to not be confused with local transitions, a different type of arrow should be used, but this is not within the scope of this thesis and a regular arrow will be used. In the examples used in this chapter the assumption that the robot has a port named *"Command"* that can receive a message called *"distance"* which holds a value, is made.

The three lines of code in figure 7.1 is the complete definition for the transition to the state run. This transition will be triggered when the




```
transitions m : Command?distance {
    (m.value > 70) -> run
}
```

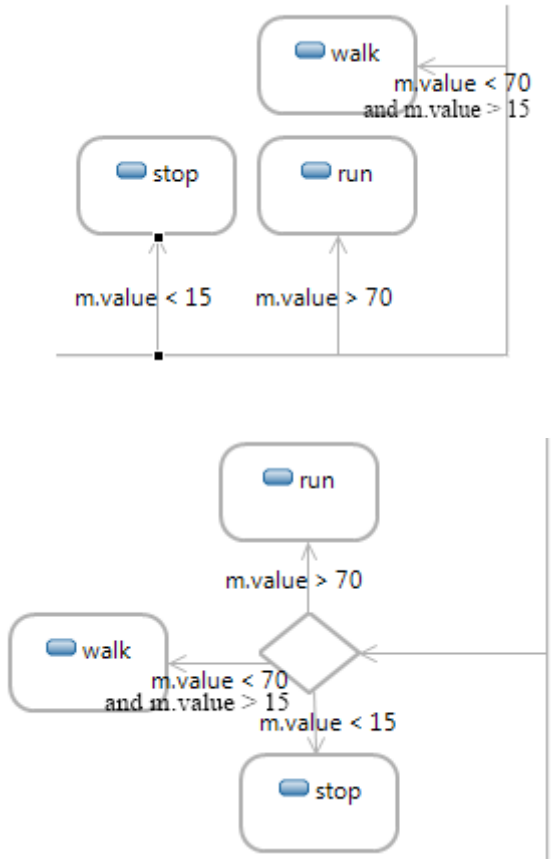
Figure 7.1: Factorized cross-cutting transition with guard, UML and ThingML code

distance is longer than 70 centimetres. This definition is placed within the state chart construct and replaces the definition of four transition in the original state machine presented in Figure 6.4 which significantly reduce the complexity of the state machine. The calculation of complexity will be explained in the next chapter. The next sections presents the practical details and semantics of factorized cross-cutting transitions and refines the definition given here.

7.2 Multiple Guards and Target States

One of the benefits of ECA rules is that they are centred around events; one rule is defined for a given event and a set of guards and actions linked to this event. This makes it easy for the programmer to understand the program reaction when a particular event is received, but makes it difficult to understand or predict the overall behaviour (after a sequence of events has been received). Being state oriented, state machines are typically less readable in this respect: what happens when an event is received mostly depends on the defined transitions and actions in the current state when the event is received. Factorized cross-cutting transitions are a combination of both paradigms, which allow defining a sort of ECA rule within a state machine, composite state or region. As such, they should be able to factorize different alternatives related to a single event.

In context of the example presented in Figure 6.4, all transitions triggered by the *"Command?distance"* message should be factorized in a single cross-cutting transition. This includes transitions to the states *run*, *walk*, and *stop*. Figure 7.2 presents how this looks in practice. A set of guards and target states can be defined within the cross-cutting transition (different actions could also be defined but is not included in this example). Using a graphical syntax this can be modelled either as separate transitions (top of Figure 7.2) or using a syntax similar to the UML *choice guard* (bottom of Figure 7.2) in order to make it explicit that the transitions share a common event. In the textual ThingML syntax, the various guards and target are contained by the transition. In the example, this single cross-cutting transition now factorizes 12 of the transitions of the model presented in Figure 6.4. Guards used in the cross-cutting transition should be written exclusive so the transitions can be deterministic.




```
transitions m : Command?distance {
    (m.value > 70) -> run
    (m.value < 70 and m.value > 15) -> walk
    (m.value < 15) -> stop
}
```

Figure 7.2: Factorized cross-cutting transition with choice guards in UML and ThingML code

7.3 Excluding States

The previously given definition of factorized cross-cutting transition distributes the transitions to all states within the scope (i.e. the state machine, composite state or region). In practice it is often the case that there are a few exceptions, which means that the transition should be distributed to ”almost” all states.

A first typical exception is the self-transition of the source states of the cross-cutting transition. By definition the source state has to be within the scope as the target state, so applying the factorized cross-cutting transition results in virtually adding a transition to itself. While this might be desirable in some case, the observation here is that it is usually not necessary. Intuitively, in many cases the cross-cutting transitions correspond to switching between modes, handling a particular type of

event or trigger an *action*. Once the mode is switched, the event is being handled or an action have been performed, the factorized cross-cutting transition does not need to be applied because it isn't always necessary to perform the same action several times in a row. For example if the robot is already walking there is no need to transit out and in to the walking state again. As an example, the model in Figure 6.4 does not include any self-transition in the states *"run"*, *"stop"* or *"walk"*. To reflect this, the default semantics of cross-cutting transitions is that they do not apply to their target state: by default no self-transitions is added. A specific attribute or keyword, *"withself"* or *"withinternal"*, can be used for cases in which the self-transitions or internal transitions should be added (the difference between the two being that a self transition actually executes *"on exit"* actions and *"on entry"* actions while an internal transition does not). The *"withself"* or *"withinternal"* keyword is to be put behind the target state in the transitions definition. A code example of this can be seen in Program 7.1 where the state *run* is allowed to transit to itself.

Program code 7.1 Factorized cross-cutting transition with self transition

```
transitions m : Command?distance {
    (m.value > 70) -> run withself
    (m.value < 70 and m.value > 15) -> walk
    (m.value < 15) -> stop
}
```

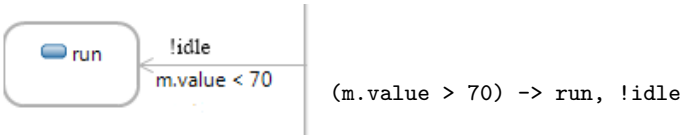
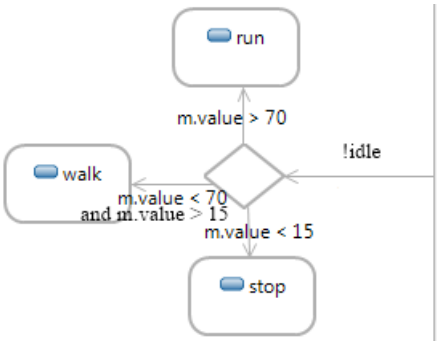


Figure 7.3: Factorized cross-cutting transition with guard and excluded state in UML and ThingML code

Another typical exceptions can be a random state that should be treated differently because of the application requirements. In the example, this is the case for the *"idle"* state, which does not include transitions to the states *"run"*, *"walk"*, *"stop"* or *"talk"*. The programmer should be able to declare in the factorized cross-cutting transition that there are some excluded states, which the transition should not trigger.

Again, there is a choice between having the transition distributed to all by default, and having an exception list or to have the transition distributed to no states except from the ones in a list. Given the main purpose of cross-cutting transition, in the approach the cross-cutting transitions are distributed to all by default and a set of exceptions can be specified. From a syntactic point of view, textually the idea is to add a list of states after the transition which correspond to the exceptions marked with a "!". In UML diagrams, the states that are excluded can be marked with a "!" and listed as seen in figure 7.3.

In case of a cross-cutting transition with multiple guards and target states, the exclusion list can either be specified as part of each target or factorized for the whole transition. Through the research it was observed that in most cases the exclusion list can be expressed for the whole transition. Figure 7.4 presents some possible concrete notations on the robot example.




```
transitions m : Command?distance {
   (m.value > 70) -> run
   (m.value < 70 and m.value > 15) -> walk
   (m.value < 15) -> stop
} !idle
```

Figure 7.4: Factorized cross-cutting transition with guard with choice point and an excluded state in UML and ThingML code

7.4 Execution semantics

The state machine semantics used in ThingML complies with the UML: when an event is received by a state machine, it is duplicated for all active regions of the state machine and passed to the inner-most current state of each region. Events can only be consumed once within each region. If the current state has a transition matching the received event it consumes the event, otherwise the state passes the event to its parent. The parent can be the state machine, a region or a composite state. At this point, if the parent has a matching cross-cutting transition it is triggered and the event is consumed, otherwise the typical execution semantics is used: at the state machine level the event is discarded, at the region level it is passed to the parent and at the composite state level it is either consumed by a transition or passed to the parent.

The proposed execution semantics for factorized cross-cutting transitions is equivalent to statically distributing the transition to the individual states except for one thing; if there already exist a transition in the state, which deals with a similar event. Distributing the transition would yield a non-deterministic choice between the two transitions, while in this case the transition defined on the state would have precedence over the cross-cutting transition, which is defined on the parent of the state. This semantics is more intuitive and consistent with the way transitions on composite states are handled.

On the example presented in Figure 6.4 the use of cross-cutting transitions allows reducing the number of transitions from 23 to 5 without

any changes in the actual behaviour of the state machine. The obtained state machine is thus significantly more compact.

7.5 ThingML Model Implementation

In order to show that factorized cross-cutting transitions actually do work and make it easier to program state machines, the ThingML language *ecore* model have been extended with the factorized cross-cutting transition suggestion shown above. The *ecore* model is a part of the Eclipse Modelling Framework (EMF). The EMF project is a modelling framework and code generation facility for building tools and other applications based on a structured data model[1]. With this model a new version of ThingML could be generated and the cross-cutting transitions tested. In figure 7.5 the addition made to the ecore file can be seen.

Together with EMFText[2] the ThingML language model is supplemented with text syntax rules that defines how the syntax rules of ThingML should look. The rules that was added to the ThingML syntax rules in order to get the new factorized cross-cutting transitions to work can be seen in program code 7.2.

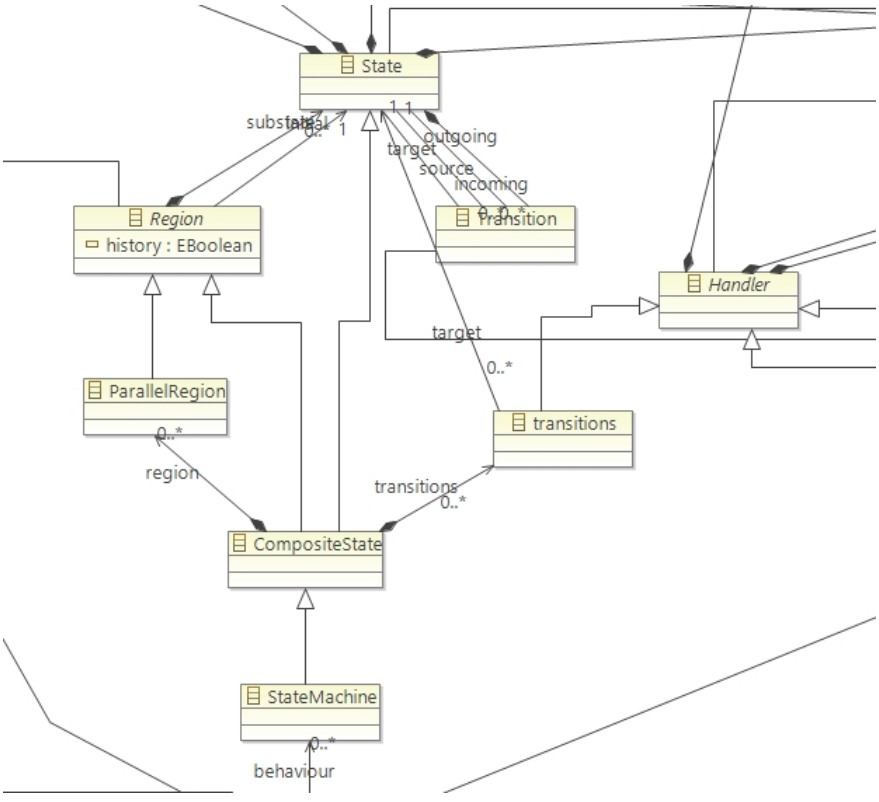


Figure 7.5: Snippet from the ThingML ecore diagram showing the transitions element

[1] http://www.eclipse.org/modeling/emf/
[2] http://www.emftext.org/index.php/EMFText

Program code 7.2 The transitions rules

```
Transitions ::= !1 "transitions" ( #1 event ) "{"

!1 (( #1 "(" guard ")" ) ( #1 name[] )? #1 "->" #1 target[]
( #1 ( "withself") ("withinternal") )* ( #1 "!" target[] )*
( !1 "action" #1 action )?
(!1 "before" #1 before)? (!1 "after" #1 after)? )*

!1 (( "(" guard ")" )? ( #1 name[] )? #1 "->" #1 target[]
( #1 ( "withself") ("withinternal") )* ( #1 "!" target[] )*
( !1 "action" #1 action )?
(!1 "before" #1 before)? (!1 "after" #1 after)? )?

"}" ( #1 "!" target[] )* ;
```

From this diagram and syntax rules it is possible to generate an editor that checkss if the program are written are valid with respect to the language rules and can generate platform specific code. The complete project is available at GitHub [14].

Chapter 8

Evaluation

This chapter evaluates the proposed approach by comparing well established metrics using different case studies realized with ”classic” state machines, as well as with the same state machines shown in Figure 6.4 enhanced with factorized cross-cutting transitions. The metrics used to compare and validate the suggested improvement are:

- the number of states (S)
- the number of transitions (T)
- the cyclomatic complexity (CC) of the state machine [26], which is calculated as follows: T-S+2
- the action complexity (AC), which is calculated by counting the number of conditional branches inside blocks of code.
- the number of ”lines of code” (LOC), which corresponds to the number of lines of the concrete (textual) syntax of the ThingML language.

8.1 ThingML robot case study

The proper way to validate the approach was to re-factor existing operational state machines (*i.e.* state machines from which fully operational code can be generated) with cross-cutting transitions. However, most state machines publicly available only remained at a very abstract level. So finally it was decided to assign the same two master-students, that already had learned ThingML and provided the feedback presented in chapter six, the following tasks:

1. Design a robot in ThingML (without cross-cutting transitions) that should change its behaviour depending on the input from several sensors, and compile a firmware for the Arduino platform.
2. Design the exact same robot with the enhanced version of ThingML (with cross-cutting transitions) and compile a similar firmware for the same platform.

The robot was equipped with an ultrasonic sensor, two light sensors (later replaced with two additional ultrasonic sensors) and three bumper sensors. It also has two DC motors to control speed and direction, speakers to output different sounds, a SOMO14D module to provide sound and several LEDs to create different light effects. All these components connected to the robot are modelled as ThingML *things* (component types) that the master-students could reuse.

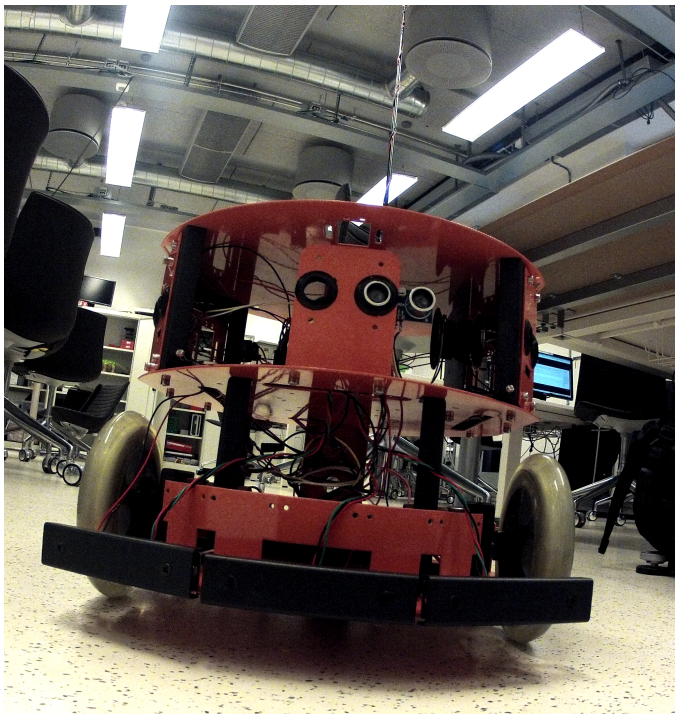


Figure 8.1: The robot used in the case study

The task that the robot, the two master-students was programming, should manage was to be able to navigate through a room without crashing, by adjusting the speed depending on the range to any object in front. The robot should also be able to choose which direction to drive depending on the light that it could sense (later changed to distance to objects on each side). Also if the robot crashed, it should manage to get away from the crash zone and make noises to signal that the programmer did not reach the goal of the task.

Before receiving their task they had not heard anything about the concept of factorized cross-cutting transitions. They were not introduced to it before they received task number two. Also, they did not know what they were tested on, or that they actually were tested at all, besides that they were supposed to make the robot work as described above. They solved the tasks individually but discussed solutions and approaches together, and they came up with two quite different solutions. A reason for why their solutions were so different, when they discussed possible solutions, might be attributed to the fact that they were rather competitive in being the first

to complete the task. Student XY who has some previous experience with state machines programmed the robot with 7 states and 1 region. Student XX had no previous experience with state machines, and programmed the robot with 8 states and 1 region. Student XY's program had an *idle* state that initializes the robot and make it ready to run. Then it goes into the *run* state that can reach each of the five other action states. A state called *crashed* can be reached from every state except itself and idle, when the robot crashes. After the *crashed* state have finished its executions, it goes back to the run state and finds out which action state to transit to. A UML diagram of student XY's program was generated from this ThingML model and can be seen in Figure 8.2. The code is available in appendix D

Student XX's program had one master state called *Start* that the program frequently falls back to in order to decide the next action to do, this occurs after the robot have crashed, stopped or changed direction. There were also two states for driving forward, one for driving fast and one for driving slow, these two states had six and seven transitions respectively.

The reason why the Start state was used to make the decision on which state to transit to after a crash, stop or turn was to avoid writing all the code for the transitions again in the different states. This also shows the need for cross-cutting transitions, handling the turning stopping and crashing. By writing all the transitions in each state the Start state would be obsolete and would make the program itself less complex. An UML state diagram was generated from student XX's program and can be seen in Figure 8.2. The code is available in appendix C.

After having programmed the robot on their own, all comments where removed from the code that they wrote and the it was formatted with uniform formatting and coding styles. Now student XY's program counted 231 lines of code, 8 states and 28 transitions, this gives a CC of **22** (28 - 8 + 2). Student XX's program counted 255 lines of code, nine states and 23 transitions, which gives a CC of **16** (23 - 9 + 2).

The students were then introduced to the new way of writing factorized cross-cutting transitions and rewrote their programs in accordance to the new rules. After rewriting, student XY's program was 188 lines long, have 7 states and 7 transitions (6 cross-cutting and 1 local). The run state from the first version had been removed since it was now obsolete. The new version of the program has a CC of **2** (7 - 7 + 2), which is a great improvement. The length of the program has also decreased with 43 lines of code, which corresponds to a decrease of 18.61 per cent of the whole program. Student XX's new program was 199 lines long, had 8 states and 8 transition giving a CC of **2** (8 - 8 + 2). The state start because the cross-cutting transitions did now handle the transitions directly from each of the states and the start state became unnecessary. The total length of the program had decreased with 56 lines of code, which corresponds to a decrease of 21.96 per cent of the program length. The diagrams made from their programs is seen in Figure 8.4 and 8.5, and the code from these programs are available in appendix F and E respectively. Note that the second region is not included in Figure 8.4 and Figure 8.5 (because it is exactly the same as in the first program) but they are of course part of the complexity calculations.

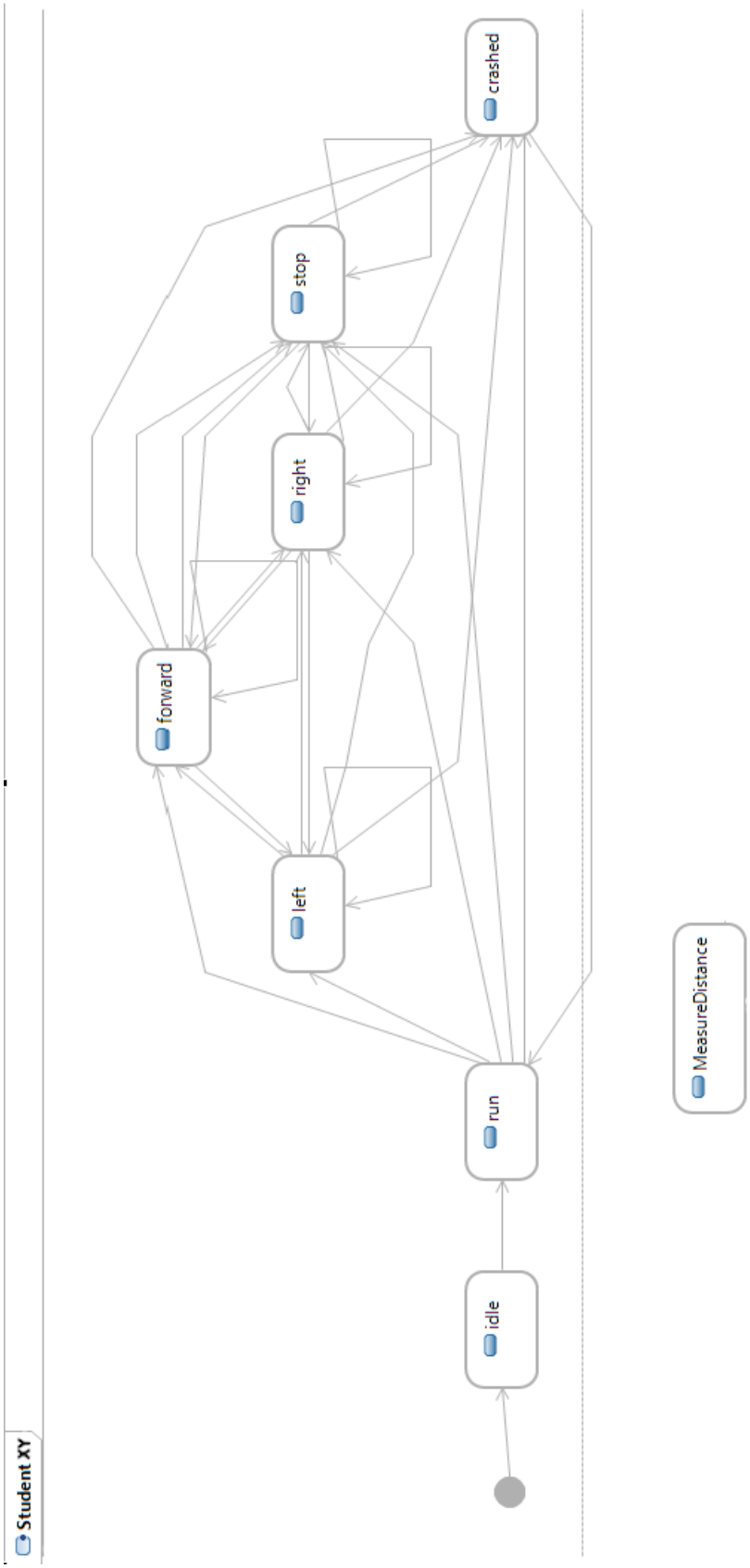


Figure 8.2: Student XY's first implementation

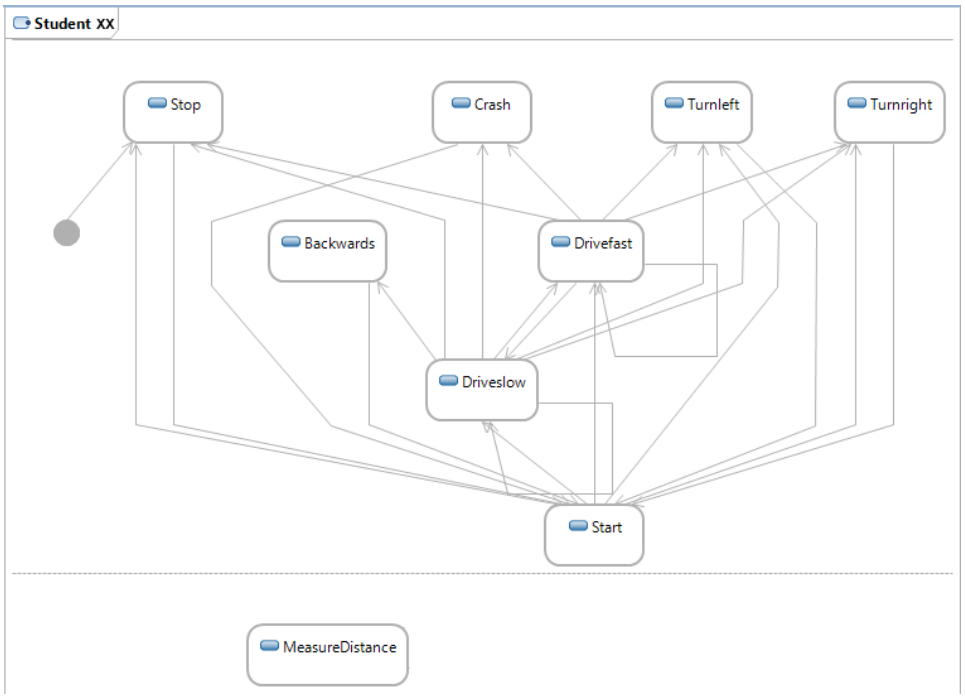


Figure 8.3: Student XX's first implementation

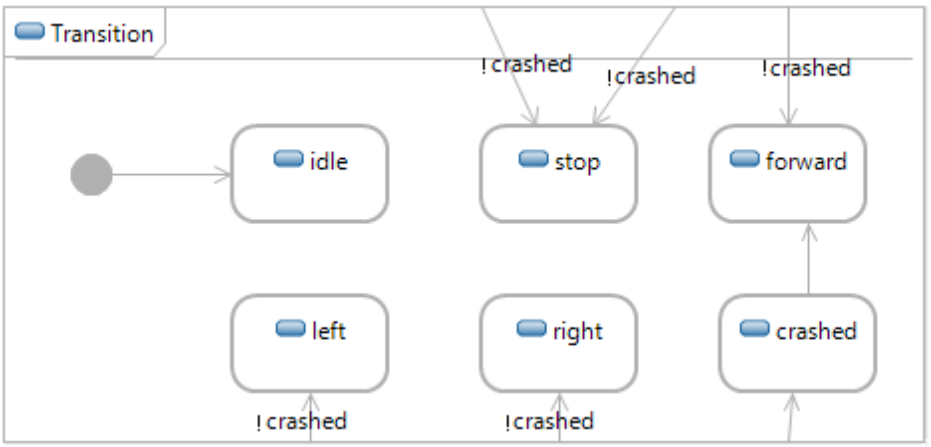


Figure 8.4: Student XY's program with factorized cross-cutting transitions

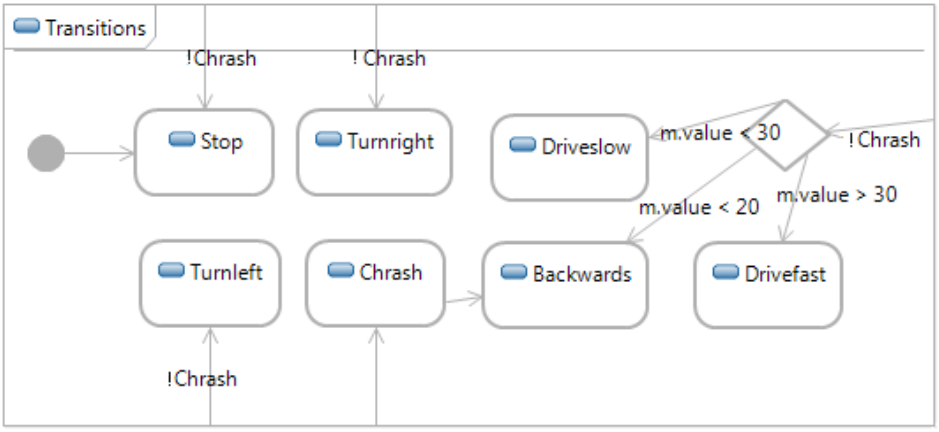


Figure 8.5: Student XX's program with factorized cross-cutting transitions

8.2 The Hypothetical Example

In chapter six there were two figures showing state machines with two and eight states (Figure 6.7 and Figure 6.8). Given that these are optimal state machines for factorized cross-cutting transitions, were each state have one transition to each of the other states, with the same event condition and the same guard. Using factorized cross-cutting transitions would give a lot of optimisation and improvement on the state machine with 8 states. In figure 8.6 a state machine with 8 states and factorized cross-cutting transitions can be seen, and if it is compared to Figure 6.8 it is easy to see the benefits in readability. The amount of code is also significantly smaller, in fact the ThingML implementation would only be 24 lines long as each transition only require 3 lines of code, given that the transitions don't have any actions bound to them selves. In the original examples with three state machines with **2**, **4** and **8** states the cyclomatic complexity would be ass follows: **2** (2-2+2), **10** (12-4+2) and **50** (56-8+2). 50 is a relatively high complexity for a state machine compared to the other state machines presented in this thesis. However after applying factorized cross-cutting transitions the complexity would be respectively **2**, **2** and **2**. This is significant results and show that it is a really big benefit in using factorized cross-cutting transitions in cases were cross-cutting transitions exists.

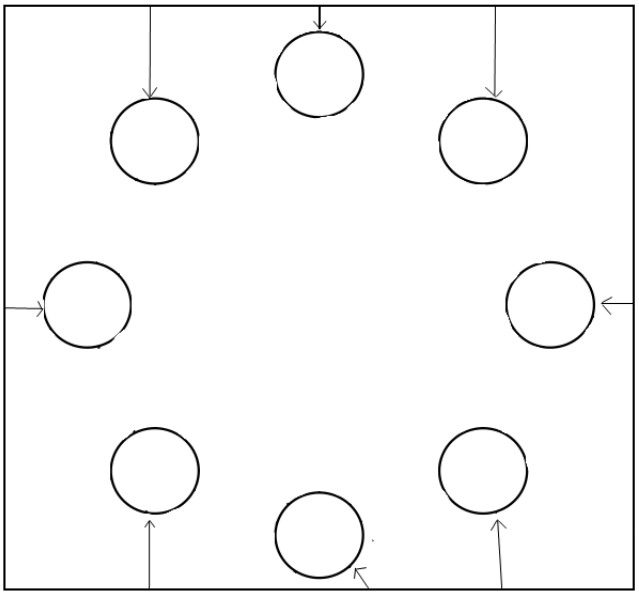


Figure 8.6: A state machine with eight states and eight factorized cross-cutting transitions

8.3 HTTP protocol state machine

As a second case study, the World Wide Web Consortium (W3C) HTTP protocol [9] is used. This protocol was one of the few ”real-life” state machines publicly available (seen in figure 8.7) that wasn't too large. It is used to assess if the approach suggested in this thesis is viable for existing state machine as well. The HTTP protocol is specified as a state machine with 6 states, five of them having an outgoing error transition to an ERROR/FAILURE exit point, which were re-factored using a cross-cutting transition. The initial complexity of the HTTP protocol, that consisted of **6** states and **14** transitions, had a CC of **10**. After applying the re-factoring, the number of transitions lowered to **10** and the CC became **6**. This shows that factorized cross-cutting transitions can be applied to existing state machines in order to lower their complexity, however at least one cross-cutting transition must be present. The results from the different examples presented in this thesis are presented in Table 8.1.

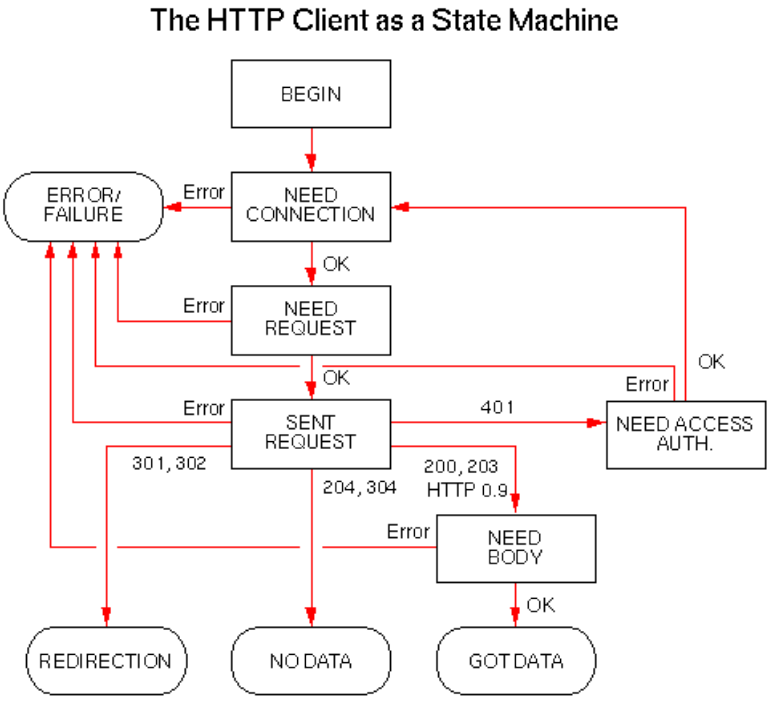


Figure 8.7: W3C HTTP protocol state machine [9]

8.4 Results and discussion

By using factorized cross-cutting transitions the complexity of the state machine can be lowered, and it is important to note that the CC is becoming lowered because the complexity is being eliminated and not hidden or abstracted away. A method to make CC calculate a lower complexity could be to use a super state that handles several transitions and hence remove some transitions and states from the diagram and therefore reducing the CC. This could be accomplished by using one or more control variables that can be used to choose what action to do within a state. In the examples presented in this article, control variables are used, they are the same for both versions of the programs developed by the students. One way to measure if a variable is used to change behaviour is to count the number of conditional tests. Student XY wrote three *if* tests in both of his programs. They were used to set the speed of the robot while going forwards and to double check if the robot actually stopped after reaching the stop state. In addition he used two guards that checked a variable in order to see which transitions the robot should transit to, these two guards are therefore counted as well. Student XX used one extra variable and four *if* tests, one of the test were used to set the speed while going backwards, and the three remaining tests were used to randomly choose between a right turn, a left turn or no turn at all after crashing. The student also used the same tests in both programs. In table 8.1 the number of extra variables and conditional tests are marked as Action Complexity (AC), and it shows that the the Action

complexity is five for student XY in both columns and 7 for student XX in both columns. These results show that using global transitions to reduce the complexity do not imply a need for extra control variables to make the program coherent with the model.

Table 8.1: Results from the examples, case study and the HTTP protocol. It shows that by inferring Factorized cross-cutting transitions the number of transitions decreased and the complexity of the state machine becomes lower. This happens without increasing the action complexity and can be seen in the students results.

	Regular Transitions				F. Cross-Cutting Transitions				
	S	T	CC	AC	LOC	S	T	CC	AC
Motivating Example	6	23	**19**	-	-60	6	5	**1**	-
Student XY	8	28	**22**	5	-43	7	7	**2**	5
Student XX	9	23	**16**	7	-56	8	8	**2**	7
HTML Protocol	6	14	**10**	-	-7	6	10	**6**	-
Example states 2	2	2	**2**	-	-	2	2	**2**	-
Example states 4	4	12	**10**	-	-	4	4	**2**	-
Example states 8	8	56	**50**	-	-	8	8	**2**	-

Chapter 9

Conclusion and Future Work

9.1 Summary and Conclusion

In the introduction of this thesis, three research topics were stated. The thesis was divided according to the three research topics which each received their own part.

The first part started out with exploring and comparing features in three microcontroller languages. It then researched what the users wants and needs in a microcontroller programming language. The results found here gave a list of features and demands that should be meet by a high level microcontroller programming language.

In the second part, ThingML was chosen as a suitable programming language that contain the features, and meet the demands identified in part one. ThingML is a new language developed at SINTEF, and because it is oriented around state machines, the language was presented in detail. ThingML was also compared to the three languages in part one in order to show how ThingML is beneficial over the other languages. Only one need, identified in the interviews in part one, could not be directly meet by ThingML, namely the need for better control structures. At the end of part two, several potential areas of improvement in ThingML were identified.

In part three, cross-cutting transitions is chosen as an important candidate for improvement in ThingML. It was chosen based on the fact that the two master-students, the developers and myself of ThingML, have all complained about the way they are handled at some point, and because handling the cross-cutting transitions are not trivial such as several of the other identified areas of improvement.

The work resulted in a new kind of transitions, which complement existing state machine models (like the UML statecharts) and aim at managing cross-cutting transitions. The factorized cross-cutting transitions provide a way to reduce complexity from the state machine structure by reducing the number of duplicated transitions and thus the amount of written code. They can be seen as ECA rules, which however have a clear scope and semantics in the context of state machines. In particular, these transitions are overridden by local transitions. This approach significantly reduces the complexity of state machines in the case where they have

”magnetic” states.

The results from the case study conducted, the inferring of factorized cross-cutting transitions in the theoretical state machines and the W3C state machine, shows that it exist potentially large benefits from using factorized cross-cutting transitions. The case study also proves that the decreased complexity from using factorized cross-cutting transitions does not increase the action complexity.

9.2 Contribution

This thesis contributes to three different fields, it can be an aid to people who want to decide which programming language they want to use for microcontroller programming by using the comparison between the languages presented in this thesis. The second area this thesis contributes to is the ThingML language, were it provides several suggestions to areas to improve in the language. One of these areas, namely the complexity caused by cross-cutting transitions was subject for an improvement suggestion. This improvement was made by inferring factorized cross-cutting transitions into the language. Factorized cross-cutting transitions can help writing more comprehensible state machines and decrease the complexity in the programs. Factorized cross-cutting transitions also makes it easier to maintain the state machine and extend it. The third area of contribution is within the general field of state machines, this is because in general all languages that are used to write or generate state machines can make use of the factorized cross-cutting transitions presented in this thesis, as is seen with the examples on UML in chapter seven.

9.3 Future work

There are several ways to continue working with the material presented in this thesis. It could take a narrow path and find ways to improve the implementation of the factorized cross-cutting transitions in ThingML. Or it can be widened out to find better ways to generalise and define factorized cross-cutting transitions to suit as many languages as possible.

The factorized cross-cutting transitions should also be tested to a larger extent on a broader audience to test the understandability of the feature. This would also mean that ThingML should try to get a larger user base so that more users can program with it and provide good answers to these questions. Further validation of the benefits from using factorized cross-cutting transitions, wit respect to decreased complexity, is also needed.

Bibliography

[1] Arduino Timed Action. http://www.arduino.cc/playground/Code/TimedAction#Example, 4 2011.

[2] Amichi Amar. *Support for Resource Constrained Microcontroller Programming by a Broad Developer Comunity*. PhD thesis, University of California, Santa Barbara, 2010.

[3] Amichi Amar and Chandra Krintz. Language support for highly resource-constrained microcontroller applications. Technical report, Media Arts and Technology and Computer Science Departments Univ. of California, Santa Barbara, 9 2010.

[4] Arduino. http://arduino.cc, 4 2011.

[5] avr lib. http://nongnu.org/avr-libc/, 4 2011.

[6] James Bailey, Alexandra Poulovassilis, and Peter T. Wood. An event-condition-action language for xml. In *Proceedings of the 11th international conference on World Wide Web*, WWW '02, pages 486–495, Honolulu, Hawaii, USA, 2002. ACM.

[7] Bokmålsordboka. http://www.nob-ordbok.uio.no/perl/ordbok.cgi?OPP=high+level+language&begge=+&ordbok=begge, 11 2011.

[8] ClicknLike. http://clicknlike.com/wi-fi-connected-lightbulbs-coming-to-smart-homes-in-2012/, 6 2011.

[9] World Wide Web Consortium. Http protocol modules as state machines., 1996. http://www.w3.org/Library/User/Architecture/HTTPFeatures.html.

[10] Mike Crang and Ian Cook. *Doing Ethnograpies*. SAGE, 2007.

[11] Diego L. de Ipia. An ECA Rule-Matching Service for Simpler Development of Reactive Applications. *Middleware 2001*, vol. 2 no. 7, 2001.

[12] Dictionary.com. Event definition in dictionary. http://dictionary.reference.com/browse/event, 5 2012.

[13] TinyOS Documentation. http://docs.tinyos.net/index.php/Main_Page, 4 2011.

[14] Kyrre Havik Eriksen, Jan-Ole Skotterud, Franck Fleurey, and Brice Morin. Thingml with factorized cross-cutting transitions. https://github.com/Kyrremann/ThingML, 2012.

[15] F. Fleurey and A. Solberg. A Domain Specific Modeling Language supporting Specification, Simulation and Execution of Dynamic Adaptive Systems. In *MODELS'09: ACM/IEEE 12th International Conference on Model-Driven Engineering Languages and Systems*, Denver, Colorado, USA, oct 2009.

[16] Franck Fleurey. Source code with comment on duplicated statements. https://github.com/ffleurey/ThingML/blob/master/org.thingml.samples/src/main/thingml_deprecated/arduino/devices/draft/statusled.thingml, 2011.

[17] Franck Fleurey. Thingml blink example source code. https://github.com/ffleurey/ThingML/blob/master/org.thingml.samples/src/main/thingml/samples/blink.thingml, 2011.

[18] Franck Fleurey, Brice Morin, and Arnor Solberg. A model-driven approach to develop adaptive firmwares. In *SEAMS'11: Proceedings of the 6th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, pages 168–177, Waikiki, Honolulu, HI, USA, 2011. ACM.

[19] Franck Fleurey, Brice Morin, Arnor Solberg, and Olivier Barais. MDE to Manage Communications with and between Resource-Constrained Systems. In *MODELS'11: ACM/IEEE 14th International Conference on Model-Driven Engineering Languages and Systems*, volume 6981/2011, page 16, Wellington, Nouvelle-Zélande, October 2011.

[20] Bent Flyvbjerg. Case study. In *The Sage Handbook of Qualitative Research*, chapter 17, pages 301–316. Thousand Oaks, CA: Sage, 4th edition edition, 2011.

[21] François Fouquet and Erwan Daubert. Kevoree. http://www.kevoree.org, 4 2012.

[22] Richard Furuta and P. Michael Kemp. Experimental evaluation of programming language features: Implications for introductory programming languages. *SIGCSE Bull.*, 11(1):18–21, January 1979.

[23] Gadgeteer. http://netmf.com/gadgeteer/, 11 2011.

[24] David Gay, Phillip Levis, David Culler, and Eric Brewerl. nesC 1.1 language reference manua, 2003.

[25] Object Management Group. Uml specifications, 2011. http://www.omg.org/spec/UML/.

[26] Mathew Hall. Complexity metrics for hierarchical state machines. In *SSBSE'11: Proceedings of the Third international conference on Search based*

software engineering, pages 76–81, Szeged, Hungary, 2011. Springer-Verlag.

[27] Runze Hao. Kevoree generator. https://github.com/ffleurey/ThingML/pull/15, 4 2012.

[28] C. A. R. Hoare. Hints on programming language design. Technical Memo AIM-224, Stanford Artificial Intelligence Laboratory, Computer Science Department, Stanford University, December 1973.

[29] Bloomberg J., Giacomi J., Mosher A., and P. Swenton-Hall. Ethnographic field methods and their relation to design. *In D. Schuler & A. Namioka (Eds.), Participatory design: principles and practices*, pages 123–155, 1993.

[30] JavaScript. http://www.w3schools.com/js/js_variables.asp, 6 2011.

[31] Bonnie Kaplan and Joseph Maxwell. Qualitative research methods for evaluating computer information systems. In *Evaluating the Organizational Impact of Healthcare Information Systems*, number 1 in Health Informatics, chapter 2, pages 30–55. Springer New York, 2005.

[32] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language, Second Edition*. Prentice Hall, 1988.

[33] Phillip Levis. *TinyOS/nesC Programming Reference Manual*, January 2006.

[34] Philip Lewis. *TinyOS Programming*, October 2006.

[35] Barr M. Real men program in C, http://www.eetimes.com/electronics-blogs/industrial-control-designline-blog/4027479/real-men-program-in-c?pagenumber=1, 8 2009.

[36] Microsoft. Modularity. http://msdn.microsoft.com/en-us/library/ff649537.aspx, 5 2012.

[37] Michael D. Myers. Qualitative research in information systems. http://www.inclentrust.org/uploadedbyfck/file/compile1997.

[38] Genneration5 Nathaniel Meyer. Finite state machine tutorial. http://www.generation5.org/content/2003/FSM_Tutorial.asp.

[39] John K. Ousterhout. Scripting: Higher level programming for the 21st century. *IEEE Computer magazine*, 31(3):23–30, 03 1998.

[40] PHP. http://www.php.net/, 6 2011.

[41] Raj Rajkumar. High level programming languages are too low level, position paper. *Department of ECE and CS, Carnegie Mellon University*, 2009.

[42] Åse Dragland. Når tingene snakker med hverandre på nettet. http://www.forskning.no/artikler/2012/juni/324126, 6 2012.

[43] Helen Sharp, Yvonne Rogers, and Jenny Preece. *Interaction Design: Beyond Human-Computer Interaction*. Wiley, 2 edition, 2007.

[44] SINTEF. Thingml, 2012. www.ThingML.org.

[45] Basic Stamp. http://www.parallax.com/tabid/295/Default.aspx, 6 2011.

[46] Arduino Threads. http://code.google.com/p/arduino-threads/, 4 2011.

[47] TinyOS. http://www.tinyos.net, 4 2011.

[48] Wiring. http://wiring.org.co/, 4 2011.

Appendix A

Interview

1. Hvilken studieretning går du?
2. Hvor langt I studieforløpet er du?
3. Hvilken erfaringer har du med mikrokontrollere?
4. Hvilken erfaringer har du med Arduino?
5. Hva er det mest avanserte du har bygd og programmert?
6. Hvor lang tid brukte du på den oppgaven?
7. Hvilken problemer støtte du på?
8. Hvordan løste du de?
9. Er det noen deler med Arduino programmeringen du mener burde ha vært gjort annerledes?
10. I den koden du har gitt til meg, hvilken deler er du mest fornøyd med?
11. I den koden du har gitt til meg, er det noen deler du ikke forstår så godt?
12. Når du skrev koden, var det ting som ikke fungerte slik som du trodde / hadde tenkt?

Appendix B

Interview Consent

Samtykkeerklæring

Forespørsel om å delta i intervju i forbindelse med en masteroppgave

Jeg er masterstudent i informatikk ved Universitetet i Oslo og holder nå på med den avsluttende masteroppgaven. Temaet for oppgaven er høynivå språk på lavnivå mikrokontrollere, og jeg skal undersøke hva som skal til for å lage et best mulig språk.

For å finne ut av dette, ønsker jeg å intervjue personer som har større eller mindre erfaring med programmering av mikrokontrollere. Spørsmålene vil dreie seg om hvordan du programmerer, feil du støter på og hvordan du fikser de, osv. Jeg vil ta notater mens vi snakker sammen. Intervjuet vil ta omtrent 15 minutter, og vi blir sammen enige om tid og sted.

Det er frivillig å være med og du har mulighet til å trekke deg når som helst underveis, uten å måtte begrunne dette nærmere. Alle innsamlede data vil være anonyme for andre enn meg. Opplysningene vil bli behandlet konfidensielt, og ingen enkeltpersoner vil kunne gjenkjennes i den ferdige oppgaven. Opplysningene anonymiseres og alle data slettes når oppgaven er ferdig, innen juli 2012.

Dersom du har lyst å være med på intervjuet, er det fint om du skriver under på den vedlagte samtykkeerklæringen og sender den til meg.

Hvis det er noe du lurer på kan du ringe meg på 90 87 94 15, eller sende en e-post til j.skotterud@gmail.com. Du kan også kontakte min veileder Franck Fleurey ved SINTEF på mail franck.fleurey@sintef.no.

Med vennlig hilsen
Jan Ole Skotterud

Samtykkeerklæring:
Jeg har mottatt informasjon om studien og ønsker å stille på intervju.

Signatur..... Telefonnummer.....

Appendix C

Original Source Code, Student XX

```
import "Devices/RangeSensor2.thingml"
import "Devices/MotorShieldDFduino.thingml"
import "Devices/LightResitorArray.thingml"
import "Devices/somo-14d.thingml"
import "Devices/Bumper.thingml"
import "../../../core/timer.thingml"
import "Devices/LightChain.thingml"

thing BigRobot includes RangeMsgs, MotorShieldMsgs, TimerMsgs, LightArrayMsgs, SoundMsgs

    required port Sounds{
     //sends r2d2
        sends chrash
        sends happy
        sends cranky
        sends set_sound //set sound with 1 2 3 or 4
        sends play_set_sound
        sends stop_sound
    }

    required port Bumper {
        receives bump
    }

    required port LightIn{
        receives forward_dir
        receives left_dir
        receives right_dir
        receives dont_know_dir
    }

    required port Motor {
```

```
        sends forward_fast
        sends forward_medium
        sends forward_slow
        sends stop
        sends backwards_fast
        sends backward_slow
        sends right
        sends left
        // sends set_motors_speed
    }

    required port Timer{
        sends timer_start
        receives timer_timeout
    }

    required port Robot {
        sends get_range
        receives range
    }

    required port Light{
        sends start_green
        sends start_blue
        sends stop_green
        sends stop_blue
        sends blink
        sends fade_blue
        sends fade_green
        sends start_crazy
        sends stop_crazy
    }

    property range : UInt16 = 0
    property rand : UInt16 = 0

statechart BigRobotImpl init Stop {

        state Start{

            on entry do
                Antenna!start () //er dette riktig?
                Light!stop_crazy ()
                Light!stop_blue ()
                Light!start_green ()
                Sounds!happy ()
            end
```

```
    transition -> Drivefast
    event LightIn?forward_dir
    guard range > 30
    transition -> Driveslow
    event  LightIn?forward_dir
    guard range < 30
    transition -> Turnleft
    event LightIn?left_dir
    transition -> Turnright
    event LightIn?right_dir
    transition -> Stop
    event LightIn?dont_know_dir
}

state Stop {

    on entry do
        Motor!stop ()
        Timer!timer_start (1000)
        Light!stop_green ()
        Light!stop_blue ()
        Light!stop_crazy ()
        Sounds!cranky ()
    end

    transition -> Start
    event Timer?timer_timeout
}

state Drivefast {

    on entry
        Motor!forward_fast ()
        Light!stop_blue ()
        Light!stop_crazy ()
        Light!start_green ()
    end

    transition -> Driveslow
    event LightIn?forward_dir
    guard range < 30
    transition -> Drivefast
    event LightIn?forward_dir
    transition -> Turnleft
    event LightIn?left_dir
    transition -> Turnright
    event LightIn?right_dir
    transition -> Stop
```

```
    event LightIn?dont_know_dir
    transition -> Chrash
    event Bumper?bump
}

state Driveslow {

    on entry do
        Motor!forward_slow ()
        Light!stop_blue ()
        Light!stop_crazy ()
        Light!start_green ()
    end

    transition -> Drivefast
    event LightIn?forward_dir
    guard range > 30
    transition -> Backwards
    event LightIn?forward_dir
    guard range < 20
    transition -> Driveslow
    event LightIn?forward_dir
    guard range < 30
    transition -> Turnleft
    event LightIn?left_dir
    transition -> Turnright
    event LightIn?right_dir
    transition -> Stop
    event LightIn?dont_know_diR
    transition -> Chrash
    event Bumper?bump
}


state Chrash {

    on entry do
        Timer!timer_start (500)
        Motor!stop ()
        Sounds!chrash ()
        Light!stop_green ()
        Light!stop_blue ()
        Light!start_crazy ()
    end

    transition -> Backwards
    event Timer?timer_timeout
}
```

```
state Backwards {

    on entry do
        Timer!timer_start (750)
        Motor!backwards_fast ()
        Light!stop_crazy ()
        Light!stop_green ()
        Light!start_blue ()

        if (range < 20)
            do
            Motor!backward_slow ()
            Light!fade_blue ()
         end

        rand = 'random(3);'
        if(rand == 1)do
            rand = 'random(2);'
            if (rand == 0) do
                Motor!left ()
            end
            if(rand == 1) do
                Motor!right ()
            end
        end
    end

    transition -> Start
    event Timer?timer_timeout
}


state Turnleft {

    on entry do
        Timer!timer_start (750)
        Motor!left ()
        Light!stop_crazy ()
        Light!stop_green ()
        Light!start_blue ()
        Sounds!set_sound (1)
        Sounds!play_set_sound ()
    end

    transition -> Start
    event Timer?timer_timeout
}
```

```
        state Turnright {

            on entry do
                Timer!timer_start (750)
                Motor!right ()
                Light!stop_crazy ()
                Light!stop_green ()
                Light!start_blue ()
                Sounds!set_sound (2)
                Sounds!play_set_sound ()
            end

           transition -> Start
           event Timer?timer_timeout
        }

        region Measure init MeasureDistance {
            state MeasureDistance{
                on entry do
                    Robot!get_range()
                end
                internal event m : Robot?range
                action do
                    range = m.cm
                    Robot!get_range()
                end
           }
        }
    }
}
```

Appendix D

Original Source Code, Student XY

```
import "Devices/RangeSensor2.thingml"
import "Devices/MotorShieldDFduino.thingml"
import "Devices/LightResitorArray.thingml"
import "Devices/somo-14d.thingml"
import "Devices/Bumper.thingml"
import "../../../core/timer.thingml"
import "Devices/LightChain.thingml"

thing BigRobot includes RangeMsgs, MotorShieldMsgs, TimerMsgs, LightArrayMsgs, SoundMsgs

    required port Sounds{
        sends r2d2
        //sends chrash
        //sends happy
        //sends cranky
        //sends set_sound //set sound with 1 2 3 or 4
        //sends play_set_sound
        sends stop_sound
    }

    required port Bumper {
        receives bump
    }

    required port LightIn{
        receives forward_dir
        receives left_dir
        receives right_dir
        receives dont_know_dir
    }

    required port Motor {
```

```
        sends forward_fast
        //sends forward_medium
        sends forward_slow
        sends stop
        //sends backwards_fast
        sends backward_slow
        sends right
        sends left
        //sends set_motors_speed
    }

    required port Timer{
        sends timer_start
        receives timer_timeout
    }

    required port Robot {
        sends get_range
        receives range
    }

    required port Light{
        sends start_green
        sends start_blue
        sends stop_green
        sends stop_blue
        sends blink
        //sends fade_blue
        //sends fade_green
        sends start_crazy
        sends stop_crazy
    }

    property range : UInt16 = 0

    statechart BigRobotImpl init idle {

        state idle {
            on entry do
                Timer!timer_start(500)
                Motor!stop()
                Light!stop_crazy()
                Sounds!stop_sound()
            end

            transition -> run
            event Timer?timer_timeout
        }
```

```
state run{
    transition -> forward
    event LightIn?forward_dir
    transition-> stop
    event LightIn?dont_know_dir
    transition -> crashed
    event Bumper?bump
    transition -> left
    event LightIn?left_dir
    transition -> right
    event LightIn?right_dir
}

state forward {
    on entry do
        if(range > 75) do
            Motor!forward_fast()
            Light!stop_green()
            Light!stop_blue()
        end

        if(range > 25 and range < 75) do
            Motor!forward_slow()
            Light!stop_blue()
            Light!start_green()
        end
    end

    on exit do
        Light!stop_green()
        Light!stop_blue()
    end

    transition-> stop
    event  LightIn?forward_sir
    guard range < 25
    transition -> forward
    event LightIn?forward_dir
    transition-> stop
    event LightIn?dont_know_dir
    transition -> crashed
    event Bumper?bump
    transition -> left
    event LightIn?left_dir
    transition -> right
    event LightIn?right_dir
 }
```

```
 state stop {
    on entry do
        Light!start_crazy()
        Sounds!r2d2 ()
        if(range < 15) do
            Motor!stop()
        end
    end

    on exit do
        Light!stop_crazy()
    end

    transition -> forward
    event LightIn?forward_dir
    transition-> stop
    event LightIn?dont_know_dir
    transition -> crashed
    event Bumper?bump
    transition -> left
    event LightIn?left_dir
    transition -> right
    event LightIn?right_dir
 }

state left {
    on entry do
        Motor!left()
    end

    transition -> forward
    event LightIn?forward_dir
    transition-> stop
    event LightIn?dont_know_dir
    transition -> crashed
    event Bumper?bump
    transition -> left
    event LightIn?left_dir
    transition -> right
    event LightIn?right_dir
}

state right {

    on entry do
        Motor!right()
    end
```

```
    transition -> forward
    event LightIn?forward_dir
    transition-> stop
    event LightIn?dont_know_dir
    transition -> crashed
    event Bumper?bump
    transition -> left
    event LightIn?left_dir
    transition -> right
    event LightIn?right_dir
  }

state crashed {

    property time : UInt16 = 0

    on entry do
        time = 400
        Timer!timer_start(time)
        Light!start_crazy()
        Motor!backward_slow()
    end

    internal event Timer?timer_timeout
    guard time == 400
    action do
        time = 500
        Timer!timer_start(time)
        Motor!right()
    end

    transition -> run
    event Timer?timer_timeout
    guard time == 500
    action do
        Motor!stop()
        Light!stop_crazy()
    end
}

region Measure init MeasureDistance {
       state MeasureDistance{
        on entry do
           Robot!get_range()
        end
        internal event m : Robot?range
        action do
```

```
                    range = m.cm
                    Robot!get_range()
                end
            }
        }
    }
}
```

Appendix E

CCT Source Code, Student XX

```
import "Devices/RangeSensor2.thingml"
import "Devices/MotorShieldDFduino.thingml"
import "Devices/LightResitorArray.thingml"
import "Devices/somo-14d.thingml"
import "Devices/Bumper.thingml"
import "../../../core/timer.thingml"
import "Devices/LightChain.thingml"

thing BigRobot includes RangeMsgs, MotorShieldMsgs, TimerMsgs, LightArrayMsgs, SoundMsgs

    required port Sounds{
        //sends r2d2
        sends chrash
        sends happy
        sends cranky
        sends set_sound //set sound with 1 2 3 or 4
        sends play_set_sound
        sends stop_sound
    }

    required port Bumper {
        receives bump
    }

    required port LightIn{
        receives forward_dir
        receives left_dir
        receives right_dir
        receives dont_know_dir
    }

    required port Motor {
        sends forward_fast
        sends forward_medium
```

```
        sends forward_slow
        sends stop
        sends backwards_fast
        sends backward_slow
        sends right
        sends left
        // sends set_motors_speed
    }

    required port Timer{
        sends timer_start
        receives timer_timeout
    }

    required port Robot {
        sends get_range
        receives range
    }

    required port Light{
        sends start_green
        sends start_blue
        sends stop_green
        sends stop_blue
        sends blink
        sends fade_blue
        sends fade_green
        sends start_crazy
        sends stop_crazy
    }

    property range : UInt16 = 0
    property rand : UInt16 = 0

    statechart BigRobotImpl init Stop {

        transitions m : LightIn?forward_dir {
            (range > 30) -> Drivefast
            (range < 30) -> Driveslow
            (range < 20) -> Backwards
        } Chrash
        transitions m : LightIn?left_dir {
            -> Turnleft, Chrash
        }
        transitions m : LightIn?right_dir {
            -> Turnright, Chrash
        }
        transitions m : LightIn?dont_know_dir {
```

```
    -> Stop, Chrash
}
transitions m : Bumper?bump {
    -> Chrash
}

state Stop {

    on entry do
        Motor!stop ()
        Light!stop_green ()
        Light!stop_blue ()
        Light!stop_crazy ()
        Sounds!cranky ()
    end
}

state Drivefast {

    on entry do
        Motor!forward_fast ()
        Light!stop_blue ()
        Light!stop_crazy ()
        Light!start_green ()
    end
}

state Driveslow {

    on entry do
        Motor!forward_slow ()
        Light!stop_blue ()
        Light!stop_crazy ()
        Light!start_green ()
    end
}

state Chrash {

    on entry do
         Timer!timer_start (500)
         Motor!stop ()
         Sounds!chrash ()
         Light!stop_green ()
         Light!stop_blue ()
         Light!start_crazy ()

    end
```

```
    transition -> Backwards
    event Timer?timer_timeout
}

state Backwards {

    on entry do
        Motor!backwards_fast ()
        Light!stop_crazy ()
        Light!stop_green ()
        Light!start_blue ()

        if (range < 20) do
            Motor!backward_slow ()
            Light!fade_blue ()
         end

        rand = 'random(3);'
        if(rand == 1) do
            rand = 'random(2);'
            if (rand == 0) do
                Motor!left ()
            end
            if(rand == 1) do
                Motor!right ()
            end
        end
    end
}

state Turnleft {

    on entry do
        Motor!left ()
        Light!stop_crazy ()
        Light!stop_green ()
        Light!start_blue ()
        Sounds!set_sound (1)
        Sounds!play_set_sound ()
    end
 }

state Turnright {

    on entry do
        Motor!right ()
        Light!stop_crazy ()
```

```
                Light!stop_green ()
                Light!start_blue ()
                Sounds!set_sound (2)
                Sounds!play_set_sound ()
            end
        }

        region Measure init MeasureDistance {
            state MeasureDistance{
                on entry do
                   Robot!get_range()
                end
                internal event m : Robot?range
                action do
                    range = m.cm
                    Robot!get_range()
                end
            }
        }
    }
}
```

Appendix F

CCT Source Code, Student XY

```
import "Devices/RangeSensor2.thingml"
import "Devices/MotorShieldDFduino.thingml"
import "Devices/LightResitorArray.thingml"
import "Devices/somo-14d.thingml"
import "Devices/Bumper.thingml"
import "../../../core/timer.thingml"
import "Devices/LightChain.thingml"

thing BigRobot includes RangeMsgs, MotorShieldMsgs, TimerMsgs, LightArrayMsgs, S

    required port Sounds {
        sends r2d2
        //sends chrash
        //sends happy
        //sends cranky
        //sends set_sound //set sound with 1 2 3 or 4
        //sends play_set_sound
        sends stop_sound
    }

    required port Bumper {
        receives bump
    }

    required port LightIn{
        receives forward_dir
        receives left_dir
        receives right_dir
        receives dont_know_dir
    }

    required port Motor {
        sends forward_fast
        //sends forward_medium
```

```
    sends forward_slow
    sends stop
    //sends backwards_fast
    sends backward_slow
    sends right
    sends left
    //sends set_motors_speed
}

required port Timer{
    sends timer_start
    receives timer_timeout
}

required port Robot {
    sends get_range
    receives range
}

required port Light {
    sends start_green
    sends start_blue
    sends stop_green
    sends stop_blue
    sends blink
    //sends fade_blue
    //sends fade_green
    sends start_crazy
    sends stop_crazy
}

property range : UInt16 = 0

statechart BigRobotImpl init idle {

    transitions LightIn?forward_dir {
        (range > 25) -> forward
        -> stop
    } crashed
    transitions LightIn?dont_know_dir {
        -> stop
    } crashed
    transitions LightIn?left_dir {
        -> left
    } crashed
    transitions LightIn?right_dir {
        -> right
    } crashed
```

```
transitions Bumper?bump {
    -> crashed
}

state idle {

    on entry do
        Motor!stop()
        Light!stop_crazy()
        Sounds!stop_sound()
    end
}

state forward {

    on entry do
        if(range > 75) do
            Motor!forward_fast()
            Light!stop_green()
            Light!stop_blue()
        end

        if(range > 25 and range < 75) do
            Motor!forward_slow()
            Light!stop_blue()
            Light!start_green()
        end
    end

    on exit do
        Light!stop_green()
        Light!stop_blue()
    end
}

state stop {

    on entry do
        Light!start_crazy()
        Sounds!r2d2 ()
        if(range < 15) do
            Motor!stop()
        end
    end

    on exit do
        Light!stop_crazy()
    end
```

```
}

state left {

    on entry do
        Motor!left()
    end
}

state right {

    on entry do
        Motor!right()
    end
}

state crashed {

    property time : UInt16 = 0

    on entry do
        time = 400
        Timer!timer_start(time)
        Light!start_crazy()
        Motor!backward_slow()
    end

    internal event Timer?timer_timeout
    guard time == 400
    action do
        time = 500
        Timer!timer_start(time)
        Motor!right()
    end

    transitions -> forward
    event Timer?timer_timeout
    guard time == 500
    action do
        Motor!stop()
        Light!stop_crazy()
    end
}

region Measure init MeasureDistance {
    state MeasureDistance{
        on entry do
            Robot!get_range()
```

```
                end
                internal event m : Robot?range
                action do
                    range = m.cm
                    Robot!get_range()
                end
            }
        }
    }
}
```

Appendix G

Arduino dynamic blink example

```
int i = 0;
void setup(){
    pinMode(13,OUTPUT);
}
void loop(){
    int del = analogRead(A0);
    if(i++ == 0){
        digitalWrite(13, HIGH);
    }else{
        digitalWrite(13, LOW);
        i--;
    }
    delay(del);
}
```